AD-A084 818 MASSACHUSETTS INST OF TECH CAMBRIDGE ARTIFICIAL INTE--ETC F/G 9/2
AUTOMATIC ANALYSIS OF THE LOGICAL STRUCTURE OF PROGRAMS,(U)
DEC 78 R C WATERS N00014-75-C-0643
UNCLASSIFIED AI-TR-492 NL

1 OF 3
AD
A084818

5✓

LEVEL II

(12)

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

ADA084818

AUTOMATIC ANALYSIS OF THE LOGICAL STRUCTURE OF PROGRAMS

RICHARD C. WATERS

80 5 27 238

WATERS

TR 92

UNCLASSIFIED (14) AI-TR-492

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER TR-492	2. GOVT ACCESSION NO. AD-A084 818	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) (6) Automatic Analysis of the Logical Structure of Programs.		5. TYPE OF REPORT & PERIOD COVERED (9) technical report,
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) (10) Richard C./Waters		8. CONTRACT OR GRANT NUMBER(s) (15) N00014-75-C-0643
9. PERFORMING ORGANIZATION NAME AND ADDRESS Artificial Intelligence Laboratory 545 Technology Square Cambridge, Massachusetts 02139		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS (12) 218
11. CONTROLLING OFFICE NAME AND ADDRESS Advanced Research Projects Agency 1400 Wilson Blvd Arlington, Virginia 22209		12. ~~REPORT DATE~~ (11) Dec~~ember~~ 1978
		13. NUMBER OF PAGES
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) Office of Naval Research Information Systems Arlington, Virginia 22217		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution of this document is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

None

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Program Analysis	Programmer's Apprentice
Program Understanding	Programming Assistant Systems
Program Verification	Understanding Loops

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This report presents a method for viewing complex programs as built up out of simpler ones. The central idea is that typical programs are built up in a small number of stereotyped ways. The method is designed to make it easier for an automatic system to work with programs. It focuses on how the primitive operations performed by a program are combined together in order to produce the actions of the program as a whole. It does not address the issue of how complex data structures are built up from simpler ones, nor the relationships

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered) between data structures and the operations performed on them.

In the method, the structure of a program is represented by a "plan" for the program. The plan is a language independent representation for the program. It abstracts away from the syntactic details of the programming language, representing control flow and data flow directly. The stereotyped methods for building up programs are referred to as plan building methods (PBMs). The most interesting PBMs are those which build up loops. They make it possible to look at a loop as a composition of stereotyped loop fragments.

The usefulness of the PBMs comes from the fact that each one has associated with it a significant amount of knowledge which can be used by an automatic system working with programs. This knowledge includes knowledge about: how to construct a program, how to analyze a program in order to determine how it could be constructed by PBMs, how to determine what the specifications for a program are, and how to verify the correctness of a program. The primary motivation behind the design of PBMs has been their use as part of an automatic analysis system which is being implemented. This system can analyze a program in order to make it easier to understand what the program does and how it achieves these results.

The PBMs and the automatic analysis system presented in this report have been developed as part of a larger reseach project. The goal of this project is to develop a system, called a Programmer's Apprentice (PA), which can assist a person who is writing a program. The purpose of the PA is to make the construction, maintenance, and modification of programs easier and more reliable. The PA is intended to be midway between an improved programming methodology which facilitates good programming style, and an automatic programming system. The intention is that the PA and a programmer will work together throughout all phases of the development and maintenance of a program. The programmer will do the hard parts of design and implementation while the PA will act as a junior partner and critic keeping track of all the details and assisting the programmer wherever possible.

This report describes research conducted at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the Laboratory's Artificial Intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-75-C-0643.

Automatic Analysis of the Logical Structure of Programs

by

Richard C. Waters

Massachusetts Institute of Technology

December 1978

This report is a revised version of a dissertation submitted to the Department of Electrical Engineering and Computer Science on August 31, 1978 in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

ABSTRACT

This report presents a method for viewing complex programs as built up out of simpler ones. The central idea is that typical programs are built up in a small number of stereotyped ways. The method is designed to make it easier for an automatic system to work with programs. It focuses on how the primitive operations performed by a program are combined together in order to produce the actions of the program as a whole. It does not address the issue of how complex data structures are built up from simpler ones, nor the relationships between data structures and the operations performed on them.

In the method, the structure of a program is represented by a "plan" for the program. The plan is a language independent representation for the program. It abstracts away from the syntactic details of the programming language, representing control flow and data flow directly. The stereotyped methods for building up programs are referred to as plan building methods (PBMs). The most interesting PBMs are those which build up loops. They make it possible to look at a loop as a composition of stereotyped loop fragments.

The usefulness of the PBMs comes from the fact that each one has associated with it a significant amount of knowledge which can be used by an automatic system working with programs. This knowledge includes knowledge about: how to construct a program, how to analyze a program in order to determine how it could be constructed by PBMs, how to determine what the specifications for a program are, and how to verify the correctness of a program. The primary motivation behind the design of the PBMs has been their use as part of an automatic analysis system which is being implemented. This system can analyze a program in order to make it easier to understand what the program does and how it achieves these results.

The PBMs and the automatic analysis system presented in this report have been developed as part of a larger research project. The goal of this project is to develop a system, called a Programmer's Apprentice (PA), which can assist a person who is writing a program. The purpose of the PA is to make the construction, maintenance, and modification of programs easier and more reliable. The PA is intended to be midway between an improved programming methodology which facilitates good programming style, and an automatic programming system. The intention is that the PA and a programmer will work together throughout all phases of the development and maintenance of a program. The programmer will do the hard parts of design and implementation while the PA will act as a junior partner and critic keeping track of all the details and assisting the programmer wherever possible.

ACKNOWLEDGMENT

The author woul l like to acknowledge the very valuable assistance of his thesis supervisor: Gerald Sussman; his readers: Carl Hewitt and Patrick Winston; his Colleagues: Charles Rich, Gerald Roylance, and Howard Shrobe; and Karen Prendergast for her assistance in preparing the illustrations.

TABLE OF CONTENTS

TABLE OF FIGURES

I. Introduction

This report (which is based on the author's PhD thesis [107]) presents a method for viewing complex programs as built up out of simpler ones. The central idea is that typical programs are built up in a small number of stereotyped ways. The method is designed to make it easier for an automatic system to work with programs. It focuses on how the primitive operations performed by a program are combined together in order to produce the actions of the program as a whole. It does not address the issue of how complex data structures are built up from simpler ones, nor the relationships between data structures and the operations performed on them.

In the method, the structure of a program is represented by a "plan" for the program. The plan is a language independent representation for the program. It abstracts away from the syntactic details of the programming language, representing control flow and data flow directly. The stereotyped methods for building up programs are referred to as plan building methods (PBMs). The most interesting PBMs are those which build up loops. They make it possible to look at a loop as a composition of stereotyped loop fragments.

The usefulness of the PBMs comes from the fact that each one has associated with it a significant amount of knowledge which can be used by an automatic system working with programs. This knowledge includes knowledge about: how to construct a program, how to analyze a program in order to determine how it could be constructed by PBMs, how to determine what the specifications for a program are, and how to verify the correctness of a program. The primary motivation behind the design of the PBMs has been their use as part of an automatic analysis system which has been implemented. This system can analyze a program in order to make it easier to understand what the program does and how it achieves these results.

The PBMs and the automatic analysis system presented in this report have been developed as part of a larger research project. The goal of this project is to develop a system, called a Programmer's Apprentice (PA), which can assist a person who is writing a program. The purpose of the PA is to make the construction, maintenance, and modification of programs easier and more reliable. The PA is designed to be midway between an improved programming methodology which facilitates good programming style, and an automatic programming system. The intention is that the PA and a programmer will work together throughout all phases of the development and maintenance of a program. The programmer will do the hard parts of design and implementation while the PA will act as a junior partner and critic keeping track of all the details and assisting the programmer wherever possible. Research on this system [79, 81, 82, 92, 105, 107] is being carried out by a group consisting of Charles Rich, Howard Shrobe, and the author at the MIT AI Laboratory, and the MIT Laboratory for Computer Science.

Chapter II gives an overview of this report. It presents the basic results without giving any more details than necessary. The remaining chapters treat each topic in depth. The exact structure of a plan is described in Chapter III, while the individual PBMs are discussed at length in Chapter IV. An experiment, discussed in Chapter V, shows that the PBMs have a wide range of applicability. Chapter VI presents a system which automatically analyzes programs in terms of PBMs. Chapter VII describes how an analysis in terms of PBMs can be used to discover what a program does. Chapter VIII discusses how PBMs could be used as the primary basis for a mini-PA system operating in the restricted domain of mathematical FORTRAN programs.

II. Overview

The research presented below is directed towards developing tools which will enable an automatic system, such as the PA, to work efficiently with programs. Two related tools are described. One is a notion of "plan". This idea, which has been developed jointly with Charles Rich and Howard Shrobe, is a representation for programs which is optimized for its utility to automatic systems. This is in contrast to a programming language which is designed to be useful to a person. The second tool is a set of PBMs. The PBMs specify how complex plans, and hence programs, are built up. The value of the PBMs is that they can be used to make it easier to understand and reason about a program.

II.1 Plans

The notion of plan presented here has been developed with several goals in mind. Some of these goals are the same as goals which are pursued when designing a programming language. Others are different, due to the differences between what is easy for a computer based system to do, and what is easy for a person to do. An important goal for a programming language, which is not a goal for plans, is the desire for a program to be easy to read when printed on paper. This forces programs to have an easily linearizable structure. Plans, on the other hand, are intended to be represented in a relational data base in a computer and can be arbitrarily graph-like.

A basic goal of plans is to represent a program completely, keeping all of the information which is necessary in order to execute the program. Part of this goal is the idea that this information should be represented in a language independent form which does not depend on the characteristics of any one programming language. Going beyond this, plans are designed to contain as much information about a program as possible. For example, a plan contains information about the specifications for the program and its parts. It also contains dependency links which show how the specifications depend on parts of the program and the way in which the parts of the program interact. This additional information serves as machine understandable annotation on the program. It specifies what the program does, and how it achieves it. The purpose of the annotation is to make it easier for an automatic system to reason about the program.

Another goal of plans is that as much information as possible should be explicit in a plan. Here, "explicit" means that the information can be directly read out of the plan without any deduction having to be done. For example, in a programming language the data flow is not explicit, but rather implicit. In order to tell where the data values created by a function are used, you have to look at the variables, assignment statements, and other mechanisms which implement data flow, and make a number of not necessarily simple deductions in order to tell where the values are used. In a plan, data flow is represented explicitly by a link between the place where a value is created and the place where it is used. The intent is to reduce the amount of deduction which needs to be done when working with a program. This goal stems from the fact that automatic systems are better at remembering things than at deduction.

A goal shared by plans and programming languages is the desire to introduce locality into a program in order to simplify the deduction which has to be done. To do this in a plan, the plan is broken up hierarchically into pieces called segments, which can be understood in isolation from each other. This is similar to the way programming languages allow a program to be broken up into subroutines. In order to guarantee that the segments can be understood in isolation, they are required to be "substitutable". Each segment is given a description of its behavior. Substitutability means that a given segment can be replaced by any other segment which has the same specifications, without changing any other aspect of a plan. This is equivalent to requiring that no feature of a plan can refer to the internal structure (i.e. something which is not in the specifications) of more than one segment at a time. The substitutability requirement implies that if you understand how, in isolation, a segment achieves its specifications, then you understand everything about the segment which is relevant to the plan.

A final goal underlying the design of plans is a requirement of naturalness from the point of view of a programmer. A plan is not necessarily intended to be directly used by a programmer. However, it is intended to be used by a system, such as the PA, which can converse with a programmer about his program. As a result, it is desirable for the concepts embodied in a plan to be natural to a programmer since these are the concepts which will be natural for the system to use.

Segments are the basic unit of a plan. Each segment takes in a set of inputs and produces a set of outputs. It has associated with it an input/output specification in the form of a set of pre-conditions, which are required to be true before the segment is executed, and a set of post-conditions, which are guaranteed to be true immediately after the segment terminates.

A segment is quite similar to a subroutine. However, it carries the basic idea of a subroutine to an extreme in two ways. First, a plan is broken up into a very large number of very simple segments. The plan is broken up into such small pieces so that the automatic system will not have to perform large steps of deduction. Subroutines written by programmers can be comparatively much larger because people are capable of larger steps of deduction.

Second, all of the relevant information about the behavior of a segment is made an explicit part of the description of the segment. For example, every value used by a segment must be an explicit input to the segment. This contrasts with a subroutine which can use devices such as free variables in order to have implicit inputs. Similarly, anything which is affected by a segment in any way must be an explicit output of the segment. Most importantly, the specifications for a segment must specify everything about the segment which any other segment depends on.

The segments in a plan are arranged hierarchically in a tree, each segment having a number of subsegments. The "outermost" segment, the one which is not a subsegment of any other segment in the plan, corresponds to the entire program which the plan is describing. The "terminal" segments, those which have no subsegments in the plan, correspond to primitive functions (such as "+", "*", or "READ") or subroutines which are being viewed as primitive. The other, "intermediate", segments correspond to logical localities within the program, as opposed to mere physical sections of the code for the program.

The behavior of a segment is achieved solely by the interaction of its subsegments. The order in which the subsegments are executed is represented by control flow links between subsegments. These links explicitly represent the ordering constraints between subsegments, independent of the mechanisms in programming languages which are used to implement control flow. The way data values are passed between subsegments is represented by data flow links between subsegments. If an output of one subsegment becomes an input to another, then a data flow link from the output to the input is used to explicitly represent this fact, independent of the programming language mechanisms which can be used to implement data flow. in order to achieve substitutability of segments both control flow and data flow are restricted so that they cannot hop over segment boundaries. This means that a control flow or data flow which originates inside a given segment must terminate inside the same segment. Together, the subsegments, control flow, and data flow describe an algorithm which can be executed to achieve the behavior of the supersegment.

Segments are divided into three classes based on the way they interact with control flow. A "straight-line" segment is a segment which receives control flow from one place, and has control flow going out from itself to one other place. a "split" is a segment which has control flow to it from one place but which has control flow leaving it going to two or more places. Splits are used to represent conditional branching. A "join" is a segment which has control flow coming into it from more than one place, and has control flow going out to one other place. Joins are used to represent the rejoining of control flow paths.

The logical structure of a program is represented in a plan by several mechanisms. Intermediate segmentation is used in order to break up the plan into a large number of small localities. Enough intermediate segments are introduced so that each segment has only a few subsegments. This reduces the problem of understanding the program as a whole to the problem of understanding a large number of small segments. It should be noted that the intermediate segmentation cannot be done arbitrarily. To be useful, the segmentation must coincide with the logical structure of the program. A key application of the PBMs described in this report is that they can be used to guide the selection of an appropriate segmentation of a plan.

As mentioned above, each segment has specifications associated with it which describe what it is supposed to do. Explicit dependency links, referred to as "justification links", are put in the plan which summarize a proof of correctness for the specifications. The justification links show how the properties of the segment follow from the properties of the subsegments and the way the subsegments are interconnected by control flow and data flow.

As an additional piece of logical annotation, each segment is identified as having an internal structure corresponding to a particular PBM. Each PBM represents a class of ways in which a segment can be constructed out of subsegments. The value of identifying the PBM which corresponds to a segment is that it makes it possible to use special case methods oriented toward that particular class of structures.

If a plan for a program only contains the minimal control flow and data flow information necessary in order to execute the program and does not have any intermediate segmentation, then it is referred to as a surface plan. If in addition to the minimal information, a plan has intermediate segmentation and if the PBM associated with each non-terminal segment is specified, then the plan is referred to as a PBM plan. If in addition to the above information, the plan contains specifications and justification links for each segment, then the plan is referred to as a complete plan.

It is very difficult to develop a complete plan for a system of programs of any size. One problem is that it is very difficult to construct complete specifications for a system of programs of any size. Another problem is that it is not possible to develop a complete plan for a program which has a bug in it, because this implies that there is some post-condition of the program which does not always hold. Further, while the program is being constructed, it is incomplete, and as a result its plan must be also. In view of all this, it is important to note that an incomplete plan is still very useful. Most of the methods which reason about a program based on its plan do not need to assume that the plan is complete. Each thing which is in the plan represents potentially useful information about the program much of which may not be at all obvious in the code for the program.

A question which has been ignored above is where all of the information in a plan is going to come from. If a programmer had to provide all this information, it would be a very tedious and error prone task because the plan is so verbose and detailed. Fortunately, at least most of the time, most of the plan can be automatically derived from the code for a program. Given the code for a program, a surface plan for the program can be straightforwardly derived. The process is essentially a process of translating from one programming language to another. Section VI.1 describes a system which performs this translation. The specifications for the primitive functions available can be entered into the system once and for all in advance. It is assumed that the programmer will provide the major portion of the overall specifications for the programs he is working on.

Determining the rest of the information which represents the logical structure of the program is more difficult. Three different kinds of systems are being developed in order to deal with this problem. Howard Shrobe [79, 92] has constructed, and is improving, a reasoning system which operates in the domain of plans. Given a segment and its subsegments, all of which have specifications, his system attempts to verify the specifications for the segment and to construct the justification links needed in the plan. However, his system is limited in its deductive abilities. Unless a plan is simple, the plan must be given intermediate segmentation, and these new segments must be given appropriate specifications, before the deductive system can be successful.

The automatic system based on PBMs developed by the author (see Section VI.2) can be used to determine a useful segmentation of a plan. The segments which result are simple enough that it is possible to automatically generate specifications for them (see Chapter VII). However, there are some inherent limitations in the specification generation process and therefore the derived specifications may not always be very useful. The specification generation process can construct justification links supporting the specifications it generates. Therefore, when the specifications generated are appropriate, the system based on PBMs can be used to construct a complete plan. In any event, the deductive system can benefit from the fact that the segmentation produced reduces the size of the deductions which must be performed.

Charles Rich [80, 82] is developing methods for recognizing instances of common algorithms and data structures in a plan. Once an algorithm is recognized, the segments corresponding to it can be given highly appropriate specifications and justification links. One problem in a recognition process like this is that there are a large number of algorithms which could match against parts of a given plan, and a large number of places in the plan where each one might match. This can lead to an explosion of competing hypotheses which swamps the recognizer. In order to try to reduce the problem of competing hypotheses, the recognition process will not be applied to a plan until after it has been segmented in accordance with the PBMs. The way the PBMs break up a program should reduce the number of hypotheses which have to be considered.

The basic notion of a plan described here was originally introduced by Charles Rich and Howard Shrobe [79]. The author described in [105, 107] a notion of a plan based on their notion which introduced the idea of a PBM and a join. All of this work built on the work of Sussman [96], Goldstein [34], and Hewitt and Smith [44]. The key ideas which are evident in this earlier work are the idea of a plan which makes the logical structure of a program explicit, and the idea of breaking a program up into a hierarchy of segments each of which is associated with a description of its behavior. The idea of expressing control flow as explicit links is used in flow charts. The work of Dennis [20] features the idea of representing data flow by means of explicit links.

II.2 Plan Building Methods

From the point of view of PBMs, a plan, and hence a program, is built up hierarchically by using a few stereotyped methods of combination. Each PBM corresponds to one of these stereotyped methods of building up a plan. If the only goal were to be able to construct programs, many different sets of PBMs would be sufficient.

The actual PBMs which have been developed are designed to meet several additional goals. The principle goal is that the PBMs should make it easier for an automatic system to perform a variety of operations on programs. Figure II-1 shows several operations which the PBMs have been designed to facilitate. The five boxes in the figure correspond to five kinds of information about a program. The arrows correspond to operations which derive one kind of information from another. The box at the top of the figure corresponds to the code for a program written in some programming language. The box below that one corresponds to a surface plan for the program (i.e. a plan without intermediate segmentation). As mentioned above, the process of translating the code for a program into a surface plan for the program is relatively straightforward. Efficiency issues aside, the reverse process referred to as coding in the figure, should also be straightforward. However, it has not been investigated in detail.

The central box in the figure corresponds to the PBM plan for the program (i.e. a plan which includes intermediate segmentation). A primary goal of the PBMs is that it should be possible to automatically analyze a surface plan in order to determine the PBM plan in a straightforward way. Going beyond that, the PBMs should be such that the analysis decomposes the plan into loosely coupled components which can be understood separately. The way the program is broken up should reveal the basic logical structure of the program by closely associating those things which are logically closely related, and separating those things which are not related.

The bottom box in the figure corresponds to the specifications which are associated with the segments in the plan. The PBMs are intended to assist in the process of automatically generating specifications for these segments. Consider a segment analyzed as built up in accordance with a PBM. The PBM should meet two key criteria. It should be easier to determine specifications for the subsegments than for the segment as a whole. Further, there should be a straightforward automatic procedure for determining the specifications of the segment based on the specifications of the subsegments and the way they are combined in accordance with the PBM. This is part of a larger goal which requires that it should be easy to develop an understanding of the whole segment based on understandings of its parts.

The box on the right of the figure corresponds to a proof of correctness for the specifications of the segments in the plan, represented in the form of justification links. Verification is the process of constructing these justification links. Each PBM is expected to assist this process in two ways. At a minimum, in those situations where the PBM can be used to automatically generate appropriate specifications for a segment, the PBM should make it easy to automatically generate justifications for these specifications. This will surely be true as long as the specification generation process is constructive. In those situations where appropriate specifications cannot be automatically generated, it would be desirable if the PBMs could still assist verification. This will be possible as long a: • cful lemmas and proof methods can be associated with each PBM.

The process of constructing a surface plan corresponding to a given PBM plan is called "construction" in the figure. In general, this is a trivial process which merely consists of deleting

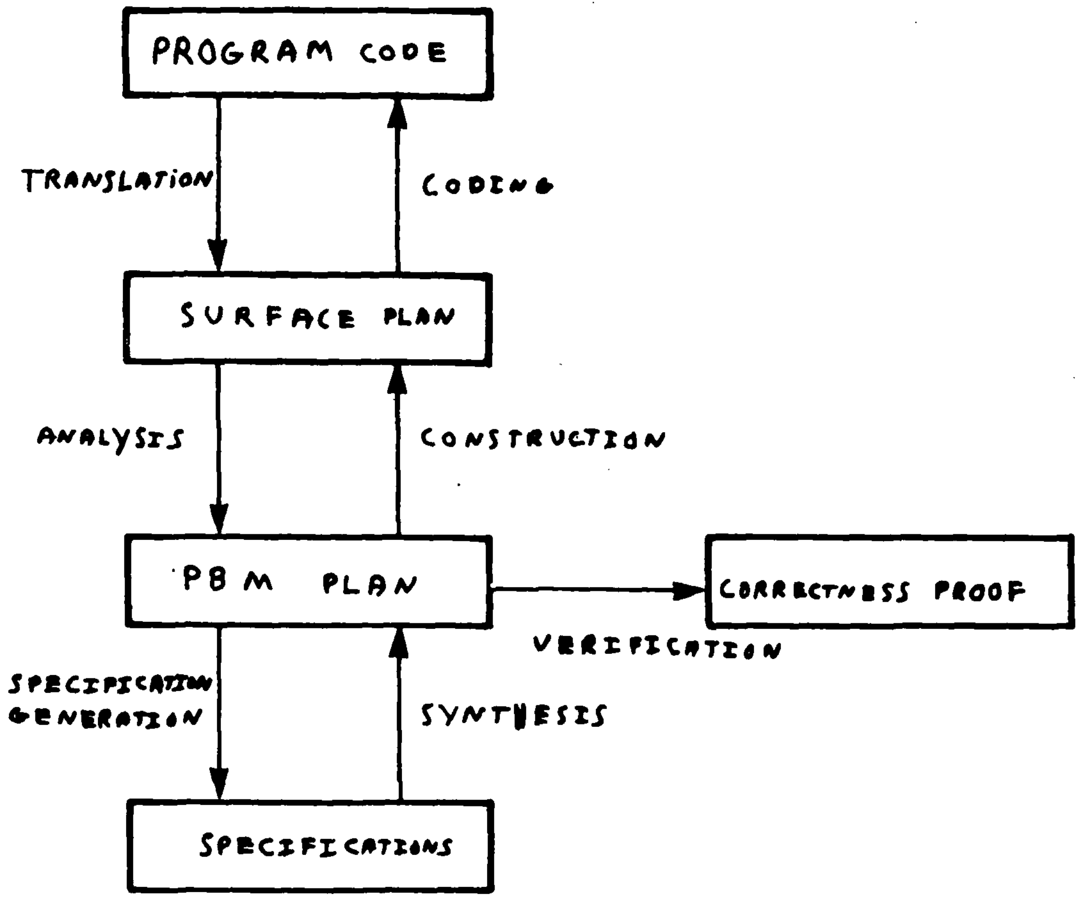


Figure II-1: Operations associated with the PBMs.

intermediate segmentation. The final operation in the figure is synthesis. This is the process of determining a PBM plan corresponding to a given overall specification. It is certainly desirable that PBMs be as helpful as possible with this task. However, this goal has not been emphasized.

To be really useful, a PBM has to have associated with it a useful package of knowledge about understanding programs in the form of knowledge about analysis, specification generation, and verification. The primary goal is that this knowledge should be usable by an automatic system, for example, to enable it to construct a complete plan for a program. As a secondary requirement, though the representation of this knowledge need not necessarily be easily understandable by a programmer, the concepts embodied in the knowledge should be familiar and natural. This requirement is motivated by the fact that like plans, the PBMs are intended to be used by an automatic system such as the PA, which will interact extensively with a programmer.

A final consideration is one of compromise. It would be nice to have a set of PBMs which satisfy all of the goals above completely, and make it possible to automatically understand any program with ease. Unfortunately, this is not possible. Human experience suggests that most programs are relatively easy to understand, while some are not. The basic notion behind the PBMs is that the reason most programs are not hard for a programmer to understand is that they are built up out of abstract parts, which he already understands, by means of methods of combination, which he already understands. The PBMs are designed to allow an automatic system to easily understand those programs which are easy to understand. Their goal is to achieve human performance, not super-human performance.

A number of researchers have worked toward goals similar to those above. One difference is that much of their work has been directed toward making it easier for people, rather than an automatic system, to work with programs. An important line of research which has been a precursor of the work reported here is based on the idea of compiling libraries of preproven algorithms and transformations as in the work of Gerhart [31, 32] and Schwartz [88]. The intent of their approach is to increase the reliability and verifiability of programs by building them up out of verified and well understood components. The PA is designed to have a library of such components. The PBMs share the feature that useful things can be proved about them in advance. However, each one is more general than a single pre-proven algorithm or transformation. They correspond to methods for combining algorithms together. An important use of the PBMs is as a first pass of analysis which breaks a program up into pieces which can be recognized as specific algorithms known to the system.

Another approach to understanding a program is to break it up along the lines of data abstractions as in the languages SIMULA-67 [12], CLU [60] and ALPHARD [112]. This is an important large scale division of the program. Sets of procedures are grouped together in data abstractions. Each group of procedures can then be understood in isolation from the rest of the program as implementing operations on a data type. This approach can be used to understand how complex data types are built up from simpler ones. However, the approach does not deal with the question of how the procedures themselves are built up. The PA is designed to work with the structure of data in a way which is similar to data abstractions. The PBMs themselves ignore the issue of the structure of data entirely. They concentrate on the procedural structure of a program.

```
      I = 1;
      Z = 0;
LOOP: IF NOT(A(I)>0) THEN GOTO SKIP;
      Z = Z+A(I);
SKIP: I = I+1;
      IF I≤N THEN GOTO LOOP;
```

Figure II-2: An example program.

The development which is most similar to PBMs has been the development of structured programming constructs. Consider the program in Figure II-2. A naive approach to looking at how this program is logically built up is based on the idea that it is constructed on a line by line basis. Given this approach, it is easy to analyze the program in order to break it up into its component parts (the six lines of the program) and the parts are indeed much easier to understand than the program as a whole. However, the problem with this approach is that it is not at all easy to discover what the program does once the parts are understood, because the effects of the lines upon each other are complex.

```
Z = 0;
DO I=1 TO N;
   IF A(I)>0
      THEN Z = Z+A(I);
END;
```

Figure II-3: The example program from Figure II-2 in structured form.

The basic structured programming constructs (composition, if-then-else, and do-while) embody a much better way of looking at how programs are built up. They indicate that a program should be viewed logically as being built up hierarchically using these structured programming constructs. Figure II-3 depicts the program in Figure II-2 analyzed in terms of the three basic structured programming constructs. The program is analyzed as being a composition of "Z=0" with an extended form of do-while which consists of counting from 1 to N in I and a body which is an if-then-else consisting of a predicate "A(I)>0", a then clause "Z=Z+A(I)", and a trivial else clause. The superiority of this approach over the naive approach outlined above is demonstrated by how much easier it is to read and understand the program in Figure II-3 then the one in Figure II-2.

When it is possible without transforming a program, this is an easy analysis to perform. This is true whether or not the program is written in a syntactically structured way. It is possible to write a program which cannot be analyzed in terms of the basic structured programming constructs, unless it is first transformed to change the topology of its control flow (for example, one which contains a loop with more than one entry point). However, a large number of programs can be directly analyzed in this way. In any case, the parts are easy to understand once they are isolated.

Let us now look at the structured programming oriented approach in the light of how easy it is to develop an understanding of the result based on understandings of its parts. First consider if-then-else. If the parts are understood, then it is easy to get an understanding of the whole. Namely, in a given situation an if-then-else either acts like the then clause, or like the else clause, depending on the value of the predicate. In Figure II-3 the if-then-else adds A(I) to Z if A(I)>0. Composition is also an easy operation to understand.

In general, these two structured programming constructs, which describe non-looping programs, meet the goals set forth above very well. Five simple PBMs (called "straight-line" PBMs) have been developed which are closely related to them. The first of these is the PBM "composition". It allows the construction of arbitrary non-branching programs. It corresponds to taking a set of segments, none of which is a split or a join, and combining them with an acyclic graph of data flow and with control flow which is consistent with that data flow. Figure II-4 shows an example of this. It does not matter whether a composition is contained on one line, or is spread over several lines in a program. The key requirement is that there not be any conditional branching.

```
Z = COS(X);
Y = A+B/2;
W = Y*Y+A;
```

Figure II-4: An example of a composition.

It is important to note that PBMs are not templates. Rather, they correspond to more general methods for building up programs. A template-like PBM could be constructed by fixing the number of subsegments, and fixing the data flow and control flow between them. The only variability which would remain would be the choice of segments to be used as the subsegments. Once they were chosen, the rest of the structure of the result would be specified by the PBM. In contrast, the number of subsegments associated with a typical PBM is flexible, and there is considerable variability allowed in the data flow and control flow in the result.

The restrictions which specify the class of structures associated with a PBM are stated in two ways. First each PBM has associated with it a set of "roles". Each role has associated with it restrictions which specify what kind of segment can fill the role. A segment can be used as a subsegment only if it can fill one of the PBM's roles. Second, restrictions are given which limit the control flow and data flow which can connect the subsegments which fill the roles in the result.

The second straight-line PBM is the PBM "expression". This is a restricted form of the PBM composition. It has the added restriction that each subsegment must be described by substitutable specifications. The specifications of a segment are said to be substitutable if each output is described by a post-condition of the form "output=F(input1,input2, ...)".

The PBM expression is introduced because it has some useful properties not shared by the more general PBM composition. One useful property is involved with the fact that the specifications for the result must not refer to any internal quantities. They can only refer to the inputs and outputs of the result. (This requirement is needed in order to ensure substitutability of segments.) With an expression this requirement is easy to meet by simply constructing a specification for the result by composing together (by means of substitution) the specifications of the subsegments. For example, this can be used to construct specifications for the program in Figure II-4. With a composition, composition of the specifications of the subsegments may not be possible, making it more difficult to derive specifications for the result.

The third straight-line PBM, "conjunction", is also a restricted form of the PBM composition. It requires that there be no data flow from one subsegment to another. It could be used to express the relationship between the first line in Figure II-4, and the last two. The key logical property of this PBM is that the subsegments have nothing whatever to do with each other. Each aspect of the behavior of the result can be traced solely to the behavior of a single subsegment.

```
IF X<10 THEN IF Y>0 THEN A=X+X;
                    ELSE A=X-Y;
        ELSE A=X*Y;
```

Figure II-5: An example of a conditional.

The fourth straight-line PBM is the PBM "predicate". It allows the combination of splits into a larger compound split. The fifth straight-line PBM, "conditional", is analogous to the structured programming construct if-then-else. It combines three types of roles: a split, a join, and some number of segments, called actions, which are neither splits nor joins. Each time the result is executed, the split selects one of the actions to perform. The program in Figure II-5 can be analyzed as a conditional which has a three way split which is a predicate built up from "X<10" and "Y>0". The properties of the PBM conditional are analogous to the properties of the structured programming construct if-then-else.

```
Z = 0;
DO I=1 TO N;
   IF A(I)>0
      THEN Z = Z+A(I);
END;
```

Figure II-6: The example program from Figure II-2 in structured form.

Returning to a discussion of the basic structured programming constructs, consider the construct do-while in the light of how easy it is to develop an understanding of the resulting program based on understandings of its parts. The body of the loop in Figure II-6 can be understood as a conditional which adds A(I) to Z if A(I)>0. Unfortunately, it is not easy to go from this to an understanding that the loop adds the sum of the positive members of the first N elements of A to the initial value of Z. It is easy to conclude this if an appropriate loop invariant can be found. However, in general, it is not easy to find such an invariant.

Another problem with the analysis in the figure is that the close relationship between the statements "Z=0" and "Z=Z+A(I)" is not made clear. In order to meet the goal of analyzing a program in such a way that things which are intimately related are closely linked together, these two statements should be put together in a single locality distinct from the rest of the loop. Having the statements separated makes it harder to understand that the program as a whole computes the sum of the positive members of the first N elements of A.

The difficulties with do-while stem from the way it looks at a loop. The body of the loop is first analyzed like any other straight-line program. This understanding of the body is then bootstrapped up to an understanding of the loop as a whole. The problem is that this bootstrapping process is far from automatic. The PBMs for loops take a different approach. They are based on the idea that the body of a loop should not be analyzed in the same way as a straight-line program. Instead, they break the loop up in order to analyze it as built up out of stereotyped loop fragments.

The lines "Z=0" and "Z=Z+_" are an example of such a fragment. The fragment computes a sum. Its fragmentary nature is indicated by the "_". The fragment needs to be connected up with something which produces a sequence of values. When this is done, it will compute the sum of these values.

```
  A           B                C               D
I=1;      IF I>N THEN                       Z = 0;
I=I+1;      GOTO EXIT;   IF A(I)>0 THEN     Z = Z+A(I);
```

Figure II-7: The example program from Figure II-6 analyzed by PBMs.

The PBMs for loops break the loop in Figure II-6 up into four fragments as shown in Figure II-7. The first fragment (A) counts up by one from one. It enumerates the sequence of integers {1,2, ...}. This fragment is part of the DO construct itself. The second fragment (B) tests the sequence of integers produced by A and stops the loop when an integer greater than N is found. This has the effect of truncating the sequence of integers to the sequence {1,2, ... ,N}. Fragment B is also part of the DO construct. The third fragment (C) operates on the truncated sequence of integers. It restricts the sequence by selecting only those integers which correspond to positive elements of the vector A. The last fragment (D) is an instance of the summation fragment mentioned above. It computes the sum of the elements of A corresponding to the integers in the restricted sequence produced by C. In the loop as a whole, the four fragments are cascaded together so that the loop computes the sum of the positive members of the first N elements of A.

The key feature of the analysis above is that it breaks the loop apart along a different dimension from the one used by an analysis in terms of do-while. There are two principle advantages to looking at a loop in this new way. First, the loop is broken up into pieces which correspond to easily understood stereotyped fragments of looping behavior. Second, the way the pieces are combined is logically equivalent to composition, which makes it easy to understand.

In order to make the idea that the fragments of the loop are composed together precise, the notion of a temporal sequence of values has been developed jointly by Howard Shrobe and the author. Given a program, such as a loop, which is repetitively executed, it can be useful to talk about the sequence of states in which some part of the program is executed, and about the sequences of values available in those states. For example, consider the statement "Z=Z+A(I)" in the loop in Figure II-6. This statement is executed in a sequence of states. There are sequences of values of I, and Z which are available in those states. These sequences are referred to as temporal sequences of values. The insight is the realization that logically, a temporal sequence of values can be treated in the same way as any aggregate data object. The concept of lazy evaluation [29, 43] uses the same insight going in the other direction. If an aggregate data object (such as a sequence of numbers) is desired, then it can be created temporally, rather than all at once, so that each piece of it is not actually created until it is needed.

Looking back at Figure II-7, fragment A produces a temporal sequence of values of I. The elements of this sequence are tested by the fragment C. Logically, the key relationship is that A creates data used by C. A is composed with C by passing this data from A to C. Viewed abstractly, it makes no difference whether this data is put into a vector which is passed all at once to C, or, as in the example, A and C are intermingled so that C can use each individual value created by A as soon as it is produced. Intermingling A and C is just an efficient way of implementing the data flow from A to C.

The key property of composition which makes it an easy process to understand is that the only interaction between two things which are composed together is that one passes data to the other. They have no other effect on each other. The loop fragments above have this vital property. Fragment A will produce a sequence of values of I counting up from one no matter what is happening in the rest of the loop. Fragment C tests the values of I it sees no matter what else is going on in the loop. As a result, each of the fragments can be understood completely in isolation from whatever loop they are being used in. (Fragment B presents a problem because it controls the number of times the other fragments are executed. In order to maintain the independence of the fragments in the face of this interaction, the descriptions of the fragments are prohibited from making any statements about the absolute number of elements in any temporal sequence of values. As a result, from the point of view of the descriptions, the fragments do not interact because their interaction cannot affect anything in the descriptions.)

Another aspect of the idea of a temporal sequence of values is that it can be used to make the notion of a loop fragment precise. The only way that the fragments are incomplete is that there are places in them where values need to be supplied (such as I in segment D). Temporal sequences of values are identified with these places. These temporal sequences are then treated as explicit inputs to the fragments as a whole. This shift of point of view is referred to as "temporal abstraction". Temporal abstraction converts a loop fragment in to a logically complete simple segment some of whose inputs just happen to be temporal sequences of values. After the abstraction step, the fragment can be treated just like any other segment. Temporal sequences of values are also associated with sequences of values produced by the fragments and are then treated as outputs from the fragment as a whole. For example, D is looked at as a segment which takes as an input a sequence of values of I and produces a sequence of values of Z. In the example in Figure II-6, the final value of Z becomes the result of the loop as a whole.

A group of PBMs have been developed which embody the idea of looking at loops as built up by composing fragments. These PBMs are called recursive PBMs because they are formulated so that they can work with arbitrary singly self recursive programs, rather than just with loops. One of these PBMs (single self recursion) corresponds to the structured programming construct do-while and is used to recognize loops which are then analyzed further by the other recursive PBMs. Three of the recursive PBMs (augmentation, filter, and termination) construct fragments. The final recursive PBM (temporal composition) combines fragments together.

The PBM augmentation constructs fragments (such as A and D in Figure II-7) which compute sequences of values. Each fragment produced is a combination of two basic roles: an initialization (such as "Z=0" in D) and a function which is repetitively executed (such as "Z=Z+A(I)" in D). Grouping these two parts together into one segment makes their close relationship clear. Based on its two roles, it is easy to generate and verify specifications for an augmentation in terms of a recurrence relation (such as "$Z_1=0 \wedge Z_{i+1}=Z_i+A(I_i)$" for D). In situations where the recurrence relation has a simple solution, better specifications can be derived (such as "$Z_i=\sum_{j=1,i-1} A(I_j)$" for D). A large number of the augmentations seen in programs correspond to simple actions such as counting, summing, or calculating a maximum. The name "augmentation" is derived form the fact that an augmentation can be added into a loop in order to augment its activities without affecting anything else the loop does.

The PBM filter constructs fragments (such as C in Figure II-7) which take in temporal sequences of values and create restricted sequences of values by filtering out some of the values. The PBM termination constructs fragments (such as B in the figure) which test sequences of values and can terminate the overall loop, causing the temporal sequences of values to be truncated. The truncation action of a termination is very much like filtering. Both filters and terminations are usually based on simple comparisons, and are easy to understand in isolation. Their action in the loop as a whole can be understood if the temporal sequences of values they test can be understood.

The PBM temporal composition combines augmentations, filters, and terminations together in order to form more complex loops. Logically, temporal composition is exactly the same as composition. the only difference is that it combines temporally abstracted fragments. As in a composition, the subsegments are combined with acyclic data flow and control flow. Some of this data flow corresponds to communicating temporal sequences of values between subsegments. Specifications for a temporal composition can be derived from the specifications of the subsegments in the same way as for a composition (see Section IV.2.1.5.2). This is very advantageous because it is easier to understand composition than any other programming structure.

The PBM temporal composition is the only PBM where the process of constructing a surface plan from a PBM plan is not trivial. The problem is that the temporal abstraction process must be undone when constructing the surface plan. The PBM temporal composition has knowledge associated with it about how to do this (see Section IV.2.1.5.1). The basic idea is that the subsegments of the temporal composition are torn apart and their subsegments intermingled in order to create an equivalent segment which does not use temporal abstraction. As a result of this, in contrast to the other PBMs which have been described, temporal composition does not just look at its subsegments as black boxes, but rather looks at their internal structure.

The process of taking apart the subsegments and intermingling them is restricted so that it preserves all of the relationships implied by the data flow and control flow between the subsegments. Thus though the data flow and control flow between the subsegments does not appear as data flow and control flow in the equivalent non-temporally abstracted segment, it completely specifies what the data flow and control flow in the equivalent segment will be. More importantly, the data flow and control flow between the subsegments accurately describes the net effect of the data flow and control flow in the equivalent segment.

The process of analyzing a loop in terms of the PBM temporal composition is essentially the inverse of the construction process described above (see Section IV.2.1.5.3). The analysis process is based on locating sections of a loop which do not have any data flow to other parts of the loop. Each such section can be removed from the loop as a fragment, and understood in isolation, because it does not affect the rest of the loop.

Unlike the straight-line PBMs, the recursive PBMs do not correspond to structured programming constructs. However, these PBMs do embody natural ideas, and many programming constructs have features in common with them. For example, there are a variety of constructs which have the feature that they separate out a sub-loop which enumerates a sequence of values from the rest of the loop which does something with them. The most common example of this is the DO statement. It makes a clear syntactic distinction between the enumeration of a sequence of integers and their use. However, in most languages, the DO statement does not correspond semantically to a sub-loop which is an augmentation because the programmer is not prohibited from assigning to the DO variable in the body of the loop. If the DO variable is modified in the body of a loop then the logical separation between the DO loop and its body is destroyed. It is no longer true that the behavior of the sub-loop is completely specified by the DO statement itself.

The MAPC function in LISP [70] is similar to the DO statement except that it enumerates the elements of a list instead of a sequence of integers. GENERATORS in ALPHARD [90] and ITERATORS in CLU [61], extend this concept to the enumeration of elements of arbitrary data types. The MAPCAR function in LISP is interesting because as well as enumerating the elements in a list, it contains a standard augmentation, whose initialization is "(SETQ X NIL)" and whose body is "(SETQ X (APPEND X RESULT))", which forms a list of the results.

The language APL [76] has several operators which operate directly on vectors in a fashion very similar to the way augmentations and filters operate on temporal sequences of values. For example, the index generator function generates a sequence of integers in a vector. The compression function filters out elements in a vector based on a bit vector. The reduction operator is similar to augmentation in that it applies an operator such as plus or times to the elements of a vector in order to compute the sum or product of all the elements in the vector.

The Language proposed by Kahn and MacQueen [51, 52] makes it possible to construct coroutine processes which interact in the same way as augmentations and filters. Each process takes in sequences of values and can produce sequences of values. The processes can be combined together into expressions which can be evaluated either sequentially, or in parallel. The processes are analogous to the APL operations on vectors, with the added feature that the vectors can be spread out in time.

II.3 Applications of PBMs

The PBMs are useful primarily because they make it easier for an automatic system to perform a variety of operations. They are intended to be used as part of a multi-faceted attack on the problem of automatically understanding a program (see the discussion of the PA in Section II.4). This report stresses the extent to which the PBMs can be the main agent used to perform these operations in order to show their utility.

II.3.1 Analysis

A system, described in Chapter VI, which automatically analyzes programs in terms of the PBMs has been implemented. The system operates in two phases. A translation phase reads in a program and converts it to a surface plan for the program. An analysis phase then analyzes the program by looking at the surface plan. The way the analysis system itself operates is not particularly novel. It is described here in order to show that the analysis process is in fact relatively straightforward. What is novel is the set of PBMs which a program is analyzed in terms of.

A translator has been written which converts FORTRAN [48] programs into surface plans. Charles Rich [79] has written a translator which converts LISP [70] programs into surface plans. After this conversion, the analysis phase of the system works on LISP programs just as easily as on FORTRAN programs.

The translator works by running over the program like an evaluator, creating control flow arcs, data flow arcs, and terminal segments as it goes. The translator has detailed knowledge of the constructs which implement data flow and control flow. It has no knowledge of what the primitive functions do except how many inputs and outputs they have.

The analysis phase operates in three steps. The first step looks primarily at the control flow arcs and analyzes the program according to the PBMs: composition, predicate, conditional, and single self recursion. This is essentially an analysis in terms of basic structured programming constructs. The analysis is done bottom up by locating minimal configurations which can be grouped together in accordance with a PBM. Whenever such a configuration is located, it is grouped together into a single segment, and the analysis continues. This process terminates when the entire program is grouped into a single segment. The process works mainly because there are only a few PBMs, and they are very different form each other. This greatly reduces the potential for conflict between hypotheses.

The first step of analysis can be compared with the system of B. Baker [3]. Her system uses graph theoretic methods in order to analyze FORTRAN programs based on their control flow in terms of basic structured programming constructs. Her system then outputs the program in a structured form. GOTOs are used in situations where an analysis in terms of the structured programming constructs is not possible without transforming the program. Her system provides further evidence that the analysis of programs in terms of structured programming constructs is a straightforward process which can be performed in a variety of ways.

After the first step of analysis, which is based largely on control flow, the system described here performs a second step of analysis which looks at the data flow and rearranges some of the groupings. The major goal of this step is to locate initializations and explicitly associate them with the segments which use them. The initialization for a segment is located by finding those segments which have data flow to and only to the segment in question. This step is necessitated by the fact that like most languages, the syntax of FORTRAN does not require the initialization for a segment to immediately precede the segment. Large parts of the program may intervene in the flow of control between the initialization and the place where its values are used.

The third and final analysis step analyzes the single self recursions discovered by the earlier steps according to the four PBMs temporal composition, augmentation, filter, and termination. This analysis is based primarily on data flow connectivity and proceeds as described in Section IV.2.1.5.3. For example, augmentations are located by finding minimal subsets of the body of a single self recursion which do not have data flow to any other part of the single self recursion. Once an augmentation is discovered, it is removed from the single self recursion and the remaining single self recursion is analyzed further.

The plan which results from this analysis is a PBM plan. The next sections discuss how this plan can be extended to a complete plan by adding pre-conditions, and post-conditions for the non-terminal segments, and logical annotation. As it stands, the plan produced is essentially a parsing of the program. It shows how the entire program can be built up from terminal segments by means of PBMs. The analysis process is limited by the fact that it is possible to construct a program which cannot be analyzed in terms of PBMs unless it is first transformed to change the topology of its data flow and control flow. Chapter V discusses an experiment which shows that a large number of programs can be directly analyzed in terms of PBMs.

II.3.2 Generating Specifications

Once a PBM plan has been constructed for a program, it can be used to help determine what the program is doing, by constructing a set of specifications for it. A system, described in Chapter VII has been implemented which automatically generates specifications based on a PBM plan. The system operates bottom up. Each PBM has knowledge associated with it which tells how to combine the specifications for the subsegments in order to generate specifications for the result. This is described in detail in the subsections of Chapter IV. Assuming that specifications for the primitive operators used in the program are known, then these specifications can be propagated up level by level in order to construct specifications for the program as a whole.

```
          PROCEDURE TR(A,N)
             Z=0;
             J=0;
             DO I=1,N;
                J=J+I;
                Z=Z+A(J);
             END;
             RETURN(Z)
          END;

   aug1        term1        aug2     aug3
   I=1;     IF I>N THEN     J=0;     Z=0;
   I=I+1;     GOTO EXIT;    J=J+I;   Z=Z+A(J);

 pre-conditions: N*(N+1)/2≤SIZE(A)
post-conditions: Z=∑j=1,N A(j*(j+1)/2)
```

Figure II-8: An example of specification generation.

Consider the program in Figure II-8. It is a loop which can be analyzed as a temporal composition of four fragments as shown. Aug1 counts up from one computing "$I_i=i$". Term1 truncates this sequence at N terminating the loop. Aug2 adds up the values of I computing "$J_i=\sum_{j=1,i-1} I_j$". Aug3 sums up values of A(J) computing "$Z_i=\sum_{j=1,i-1} A(J_j)$". It is interesting to note that aug1, aug2, and aug3 are all cases of fundamentally the same summation fragment. Specifications for the loop as a whole can be derived by composing together specifications for the four fragments. It can be seen that "$J_i=\sum_{j=1,i-1} j = (i-1)*i/2$", and therefore the final value of Z computed is "$Z=\sum_{j=1,N} A(j*(j+1)/2)$". The requirement that the values of J be in the bounds of the vector A is reflected as the pre-condition "$N*(N+1)/2 \leq SIZE(A)$" of the program as a whole.

The specifications produced are a quite reasonable description of what the program does. However, they are not the best specifications possible. There are two basic problems with the automatically generated specifications: They are too general, and they do not take advantage of any special concepts which are appropriate for the program.

```
 pre-conditions: TRIANGULAR-MATRIX(A,N)
post-conditions: Z=TRACE(A)
```

Figure II-9: Improved specifications for the example program.

The improved specifications in Figure II-9 reflect the fact that the program is intended to compute the trace of an NXN triangular matrix stored in an efficient way in vector A. The automatically generated specifications are too general in that they specify that the only necessary relationship between A and N is that N*(N+1)/2 be less than the size of A. The programmer intended that N*(N+1)/2 must be exactly the size of A. In addition, the automatically generated specifications do not use knowledge of the fact that a triangular matrix is stored efficiently by storing only the elements on and below the diagonal in row order nor the fact that the sum of the diagonal elements of a matrix is the trace of the matrix.

The problem of excess generality is a fundamental one. There is nothing in the program which indicates anything more specific than the specifications in Figure II-8. The specifications in Figure II-9 are actually less accurate (though more appropriate). In order to determine the more restricted specifications, the programmer's intentions have to be taken into consideration. Plans and the PA are able to deal directly with the programmer's intentions. However, this requires annotation by the programmer, and is not revealed by an analysis of the code for a program in isolation. The problem of using the appropriate higher level concepts also depends on the programmer's intentions.

The two problems above limit the situations in which the kind of automatic generation of specifications described here is liable to be useful. The specifications are constructed by pushing information from the specifications of the terminal segments up to higher levels. As specifications are built up at higher and higher levels, problems due to lack of higher level knowledge, and excessive generality, mount up until the specifications become more and more inappropriate. As long as the generation process extends only across three, four, or five levels, the specifications which result are not unreasonable. However, going further than this leads to specifications which are not liable to be useful.

The PA is designed to utilize the kind of specification generation described here. However, it also uses comments provided by the programmer so that specifications do not have to be generated across more than a few levels. The automatic generation is used essentially to fill in the gaps in what the programmer says, and to link what he says to the program.

II.3.3 Verification

The most fundamental impact of the PBMs on verification is that they break up a program into a hierarchy of segments within segments. This structure can be used as the underlying structure of a proof of correctness, by determining specifications for each segment and then verifying these specifications. With each subproof, if the generated specifications are appropriate, then the PBMs can do all the work since justifications for the generated specifications are a natural byproduct of the way they are derived. A problem with all this is that the specifications generated, though accurate, may be different from the specifications the programmer desires to show are correct. In this situation, the method which is used to generate a proof can benefit from the fact that it only has to work on the single subproof which has been singled out by the PBMs, rather than on the program as a whole.

```
X = 0;
Y = 0;
L = 11;
DO K=1,10;
   IF C(K)>0 THEN DO;
      X = X+A(K);
      Y = Y+A(K)*A(L);
   END;
   L = L+1;
END;
```

the program is claimed to compute

$X=\sum_{\{i \mid i\in\{1,\ \ldots\ ,10\} \wedge C(i)>0\}} A(i) \wedge$
$Y=\sum_{\{i \mid i\in\{1,\ \ldots\ ,10\} \wedge C(i)>0\}} A(i)*A(i+10)$

assertions summarizing the actions of the body of the loop

$L=L'+1 \wedge (C(K')>0 \rightarrow (X=X'+A(K') \wedge Y=Y'+A(K')*A(L'))) \wedge$
$(C(K')\leq 0 \rightarrow (X=X' \wedge Y=Y')) \wedge K=K'+1$

a loop invariant

$0\leq K\leq 11 \wedge L=K+10 \wedge$
$X=\sum_{\{i \mid i\in\{1,\ \ldots\ ,K\} \wedge C(i)>0\}} A(i) \wedge$
$Y=\sum_{\{i \mid i\in\{1,\ \ldots\ ,K\} \wedge C(i)>0\}} A(i)*A(i+10)$

Figure II-10: Steps in a proof by the standard single invariant method.

The impact of the PBMs on the structure of a correctness proof is greatest with regard to loops. The process of temporal abstraction leads to a proof which is fundamentally different in form from the standard approach of using a single loop invariant. Consider the program in Figure II-10 and how it would be verified by using a single loop invariant as originally introduced by Floyd and Hoare [26, 45]. Assertions are passed over the body of the loop in order to develop a statement of what the body does. Then an appropriate loop invariant is developed. Finally, the statement of what the body does is used to prove the invariance of the loop invariant, and the loop invariant is used to verify the specifications of the loop.

One of the most difficult steps in a proof of this form is the determination of an appropriate loop invariant. Considerable research has been done on ways to automatically develop invariants. Much of this work centers around heuristic methods which can be used to guide a search for an invariant [33, 55, 110]. Some of it is oriented toward directly deriving invariants for specific classes of loops [8, 9, 72]. The work of Basu and Misra [8, 9] is particularly interesting. They analyze the mathematical properties of a loop in order to directly derive an appropriate loop invariant for certain classes of loops.

PBM	section of the program	the sequence of values it produces

aug.	`K = 1;`
	`K = K+1;`
		$1 \le i$
		$K_i = i$

term.	`IF K>10`
	`THEN GOTO EXIT:`
		$1 \le i \le 10$
		$K_i = i$

aug.	`L = 11;`
	`L = L+1;`
		$1 \le i \le 10$
		$L_i = i+10$

filter	`IF C(K)>0 THEN`	the above sequences are restricted to the elements which correspond with positive elements of C(K)

aug.	`X = 0;`
	`X = X+A(K);`
		$1 \le i \le 10$
		$X_i = \sum_{(j \mid j \in \{1, \ldots, i\} \wedge C(j)>0)} A(j)$

aug.	`Y = 0;`
	`Y = Y+A(K)*A(L);`
		$1 \le i \le 10$
		$Y_i = \sum_{(j \mid j \in \{1, \ldots, i\} \wedge C(j)>0)} A(j)*A(j+10)$

Figure II-11: Steps in a proof based on PBM analysis.

Figure II-11 shows how the program in Figure II-10 would be analyzed by PBMs, and how this leads to a proof of correctness. The program is divided into six parts: an augmentation which enumerates integers in K, a termination which truncates this to the sequence of integers from 1 to 10 in K, an augmentation which enumerates the integers from 11 to 20 in L, a filter which restricts these temporal sequences of values by selecting only those elements which correspond to positive values of C(K), an augmentation which computes the sum of the indicated elements of A, and an augmentation which computes the sum of the products of the indicated elements of A. This decomposition leads to a style of proof based on composition which is very different from the proof in Figure II-10. First several lemmas are proved. Each lemma summarizes the actions of one of the parts of the program. Second, the lemmas are combined by composition in order to yield the desired result.

The problem of finding the loop invariant is dealt with by breaking it up into pieces. No invariant is needed in order to verify the program as a whole. Rather, each of the proofs of the lemmas requires an invariant. However, each of these proofs is so simple that it is easy to determine what the invariant should be by the methods of Basu and Misra, if not by simple recognition. It should be noted that not all programs can be decomposed by PBMs as nicely as the one in the example. There is no limit to the complexity of the pieces which result. Therefore, it may be very difficult to determine the invariant needed to prove one of the lemmas needed. Even in this situation, the PBM analysis is useful because it determines the parts of the invariant as a whole which are easy and separates them from the parts which are difficult. PBM analysis is not a uniform procedure which will determine the invariant for any loop; rather, it simplifies the problems involved with finding most of the invariant for most loops.

The fundamental difference between the form of the proof engendered by PBM analysis and the form of the proof resulting from the single invariant method becomes apparent when the proof is used for something other than giving a yes/no answer to the question of whether or not the program is correct. For example, suppose that the program were incorrect and that therefore the proof failed. The PBM proof is broken up into a sequence of steps which are directly linked to parts of the program. If the failure of the proof can be localized to one of the steps of the proof, then the bug in the program can be localized to the corresponding part of the program. The failure of a proof based on a single loop invariant does not lend itself to this kind of analysis.

The same kind of difference appears in a variety of other tasks. For example, the PBM proof can be used to help explain how the program works because it indicates what parts of the program contribute to what parts of the specifications. The analysis according to PBMs, and the resulting proof of correctness, are specifically designed to reveal the logical structure of a program.

II.3.4 PBMs, Synthesis, and Programming Languages

Synthesizing a program given its specification is a difficult task in general. However, in certain situations, it is easy. The contribution of the PBMs to synthesis is that they increase the number of situations where synthesis is easy. Synthesis using PBMs operates in three steps. First, based on a specification, a PBM plan is constructed which has the given specification. This step is accomplished by using the knowledge associated with the PBMs in order to reverse the process of generating specifications. Second, using the information about how to combine the subsegments of each PBM, a surface plan is constructed. Third, a program is constructed in some programming language based on the surface plan. The last two steps are straightforward. The first step is the critical one.

The question of whether or not a PBM plan can be easily synthesized to meet a given specification boils down to a question of what the form of the specification is. The PBM plan will be easy to synthesize only if the specification has the same form as specifications which are automatically generated. For example, it is easy to synthesize a plan for a program which is required to calculate Z such that "Z=(-B-SQRT(B*B-4*A*C))/2*A", or such that "Z=(IF SQRT(B*B-4*A*C)≥0 THEN (-B-SQRT(B*B-4*A*C))/2*A IF SQRT(B*B-4*A*C)<0 THEN -B/2*A)". In contrast, it is not easy to synthesis a program computing Z such that "A*Z*Z+B*Z+C=0" or such that "∃Y(Z=REAL-PART(Y) ∧ A*Y*Y+B*Y+C=0)". Synthesizing programs which match the simple specifications above is so easy that it is often not thought of as synthesis. A compiler performs this kind of simple synthesis.

The recursive PBMs suggest that a larger class of specifications is just as easy to synthesize programs for. For example, specifications with requirements such as: "$Z=\Sigma_{i=1,10}A(i)$", "$Z=\Sigma_{\{i|\ 1\le i\le N \land C(i)>0\}} B(i)*B(i)$", "Z=number-of(i| 1≤i≤N ∧ C(i)>0)", or any of the other post-conditions in specifications generated for temporal compositions (see Section IV.2.1.5.2). A simple specification could also be stated directly as a recurrence relation such as "$Z=X_{10} \land X_1=0 \land X_i=X_{i-1}*A(i)+X_{i-1}$". It is straightforward to synthesize programs which implement specifications like the ones above. It should be noted, however, that the programs produced may be far from optimal. Efficiency problems also appear when synthesizing code for ordinary expressions.

Several points should be made about the kind of synthesis described above. The class of specifications being talked about is very restricted. It does not include non-algorithmic specifications, and fails to include many algorithmic ones, for example, those which refer to functions not known to the system. The kind of synthesis described here is only useful in situations where it is reasonable to construct the appropriate kind of specifications. This really returns to the topic of programming languages. The simple specifications can be looked at as a programming language. Looked at from this point of view, the key question is when these simple specifications are either simpler, or clearer than standard programming languages for specifying a program. With regard to the straight-line PBMs, the simple specifications are basically the same as standard programming languages. The example specifications associated with recursive PBMs shown above seem to be simpler and clearer than standard programming languages for describing the programs they describe. As a result, they might be a useful addition to a programming language. However, the generated specifications for some recursive programs are very unwieldy and unclear. For these programs, standard languages are superior to specifications like those which are automatically generated.

Another approach to introducing the recursive PBMs into a programming language is to try and create structured programming constructs corresponding to them. Current programming languages lack structured notations for multiple augmentations and terminations. Each structured construct in a language plays two roles. It provides a tool for the programmer to use when he is constructing the program in the first place. More importantly, the use of the construct makes the program more readable, because it makes the logical structure of the program more apparent. In this role, it enhances the value of the code for the program as documentation.

One important aspect of any construct is how it will look when a program is printed on paper. This is particularly important in its role as documentation. Virtually all structured constructs are expressed using syntactically nested notations. An example of a syntactically nested notation is the if-then-else statement: "IF pred THEN statement1; ELSE statement2;". This specifies how pred, statement1, and statement2 are to be combined without referring to the internal structure of any of them. The construct treats them like black boxes. Syntactically nested constructs have several very nice features. In particular, they give the program a tree like structure and provide a lot of locality. All of the information about how the pieces will fit together is contained solely in the construct itself, and each piece can be understood in isolation.

One reason that temporal composition augmentation, filtering, and termination have not appeared in full form in languages may be the fact that they do not lend themselves to being expressed in syntactically nested notations. A nice syntactically nested construct such as (TEMPORAL-COMPOSITION AUG1 AUG2) will not work because it does not fully specify how the two augmentations are to be combined together. Viewed more abstractly, it does not specify what temporal sequences of values the augmentations will receive. The problem is that the augmentations cannot be looked at simply as black boxes during the process of temporal composition. Using current programming languages, a programmer specifies how two augmentations are combined, by combining them into one loop. This does an effective job of specifying the necessary information, but it does not make the logical structure of the resulting loop apparent in the code for the program.

Constructs such as DO and MAPCAR make a compromise between the desire to have a syntactically nested construct, and the fact that the pieces to be combined cannot be looked at as black boxes. With both constructs, a basic augmentation which enumerates some values, and the way a second augmentation will be combined with it are specified by conventions in the underlying semantics of the construct, and are not specified by the programmer. It is not clear whether this approach of using conventions to specify the combination of augmentations could be extended in order to handle multiple augmentations with a nested construction which would serve as good documentation.

One way to introduce augmentations and filters into a programming language would be to move up to a more abstract level which talks directly in terms of temporal sequences of values. At this level, augmentations and filters can simply be composed together and can therefore be expressed easily in a syntactically nested way. This would lead to a language which had constructs comparable to the operators in APL. This is the approach taken by Kahn and MacQueen [51, 52]. Their language is capable of expressing a program directly as a set of coroutines operating on vectors which may be spread out in time. In order to get the significant savings in time and space which result from implementing augmentations and filters with loops operating on temporal sequences of values rather than by functions operating on actual vectors of values, the language would then have to have a smart compiler. The compiler would have to know how to combine the pieces together into a loop.

A different approach to introducing the recursive PBMs into a programming language could be based on an interactive program editing environment. In such an environment, the programmer could develop a program in a structured way by issuing commands in terms of PBMs. In response to a command such as "add this augmentation into that loop operating on that temporal sequence" the editor would modify the loop to include the additional augmentation. The programmer would have the added advantage of being able to edit the resulting loop in order to increase its efficiency by promoting sharing and the like if necessary.

The main problem with this approach is that it does not contribute a construct which aids in documenting the code. However, the fact that the logical structure of the program cannot be printed out in a nice tree-like manner does not prevent the editing system itself from remembering what the logical structure of the program is (for example, by using a plan for the program). This could be used to the programmer's advantage if the editing system itself could use this knowledge of the structure of the program in order to aid the programmer. This is the approach taken by the system described in the next section.

II.4 The Programmer's Apprentice

The PBMs discussed in this report were developed in the context of the design of a system which can assist a person who is working on a program. The purpose of the system, which has been called a programmer's apprentice (PA), is to make the construction, maintenance, and modification of programs easier and more reliable. The intention is that the PA and a programmer will work together throughout all phases of the development and maintenance of a program. The programmer will do the hard parts of design and implementation while the PA will act as a junior partner and critic keeping track of all the details and assisting the programmer whenever possible.

The PA is designed to perform a variety of tasks. Underlying all of these tasks is the ability to understand the program which is being worked on in a way which allows it to effectively communicate with the programmer about the program. One of the primary roles of the PA is to serve as continuing in depth documentation for the program. It can describe the structure of the program and answer questions about it. The PA can aid in verifying a program. As part of understanding a program, the PA knows why it works. This is the backbone of a proof of correctness. The PA can aid in debugging a program. While the program is being developed, the PA is developing an understanding of why it works. The PA detects bugs in the program by detecting inconsistencies in this understanding. It can then aid in localizing the bug by using its understanding of what parts of the program are causing the inconsistency. The PA can help assess the effects of a modification. Since it knows the logical dependencies in the program, it can determine what parts of the program can be affected by a proposed change, and can draw some conclusions about these effects.

The PA is designed so that it has so much explicit information about the program being worked on that it can perform each of the above tasks using only small steps of deduction. This being the case, *the primary effort in designing the PA has been directed toward determining what kinds of information it should have about a program, and how it can acquire this information.* The PA has knowledge of the structure of the data objects used by a program organized in a manner similar to data abstractions. In addition, it has knowledge of the logical structure of the program itself. This information is embodied in a plan for the program.

Consider how the PA can perform the tasks described above given that it can develop a plan for a program. In its role as documentation, the PA merely reports out information stored in the plan. The justification links directly record the backbone of a proof of correctness. Bugs are detected as contradictions among the justification links. Localizing bugs and assessing modifications are both accomplished by looking at the justification links in order to determine what depends on what.

The pivotal activity of the PA is the process of acquiring the knowledge which is represented by the plan for a program. PBMs play an important role in this process. All of the information in the plan must be either known to the PA in advance, supplied by the programmer, or derived by the PA. The PA knows about the primitive operations, and about a variety of common data structures and algorithms. The programmer supplies the specifications and design for the program to be written. The PA and the programmer cooperate when working in the areas between these two extremes. In one major mode of interaction, the programmer writes code, and the PA uses the PBMs in order to break the code up into pieces which it knows about. It can then use the knowledge associated with each PBM in order to build up a complete plan. The PA has a reasoning system which performs the deductions required by other parts of the system. The major role of the PBMs is to break problems up so that the deductions which the reasoning system is requested to perform are never excessively difficult.

Another mode of interaction which is possible is that the programmer can talk directly with the PA about the evolving plan. The PBMs are useful as a vocabulary which can be used by the programmer and the PA when talking about the plan and the logical structure of the program. The programmer can also refer directly to the algorithms and data structures which the PA knows about. Given a complete plan, it is easy for the PA to produce code.

In order to highlight the way PBMs are used in the PA, Chapter VIII describes a mini-PA which would operate in the restricted domain of mathematical FORTRAN programs. In this domain, the only data structures used are numbers and vectors. As a result, the parts of the PA dealing with data structures can be deemphasized. Similarly, there are not very many basic algorithms used. Most of the computation corresponds to mathematical formulas. As a result the PA does not need very much knowledge of common algorithms. Most importantly, in the restricted domain, a considerable amount can be done with only very simple deductions. This makes it possible to design a mini-PA the primary component of which is the PBMs.

As mentioned above, the research on the PA is a three person project. Charles Rich and Howard Shrobe laid out the basic goals for the PA and the basic description of what constitutes a plan [79]. The author outlined a PA system which would operate in the restricted domain of FORTRAN programs [105]. Since that time, research has continued in parallel in three main areas. Howard Shrobe [91, 92] has implemented a prototype reasoning system which operates in the context of plans for programs. It is designed to be able to perform the many deductions needed by the PA, and to aid in deriving the justification links in the plan for a program. Charles Rich [80, 82] is working on the problem of how knowledge of common data structures and algorithms should be represented in the PA, and what things it should know about a particular programming domain. He is also investigating how the PA can recognize instances in which a programmer has used data structures and algorithms it knows about. The author [106, 107, 108] has concentrated on studying how the logical structure of a program is built up. This has led to the PBMs. They are important in the context of the PA because they enable the PA to understand a program by breaking the program up into pieces which it knows about and then combining its understandings of the pieces into an understanding of the program as a whole.

II.5 Relationship To Other Work

This section discusses the relationship of the PA as a whole, and plans and PBMs in particular, to other research. The PA is intermediate between an improved programming methodology and an automatic programming or verification system. It shares many features with both ends of this spectrum.

The development of improved methods of programming has been one of the most successful approaches to increasing programmer productivity and program reliability. Starting from assembler language programming, an important step was the development of high level languages such as FORTRAN, PL/I, and ALGOL. These languages make readily available commonly used operations, common methods for combining operations together, and methods for controlling the complexity of the resulting program. For example, they have mechanisms for writing expressions, calling subroutines with arguments, and structured constructs such as if-then-else and do-while. Another important idea introduced by high level languages is the idea of a data type. One of the important impacts of these ideas is that they make it easier to read and understand a program.

There are a variety of ways in which the basic ideas above have been extended. One is the design of very high level languages which incorporate a great deal of knowledge about a particular programming domain. An example of this is the language BDL [41] which can be used to write business data processing programs. BDL makes many assumptions about what the form of the resulting program will be, and has many constructions which make it very easy to write a program of that form. Such a language is very helpful when writing programs in the appropriate domain, but can be an impediment when writing a program sufficiently far outside that domain. The natural culmination of the trend of putting more and more domain specific knowledge in a programming language is a customizer. With a customizer, the programmer's task is reduced to determining whether or not the program he wants is one of the relatively few programs the customizer can produce. In order to maintain applicability to a wide range of programs, the PA is not oriented toward any particular domain. However, the PA has knowledge of useful algorithms in its plan library. The more knowledge it has about a given domain the more useful it will be in that domain.

Another way in which the ideas in the basic high level languages have been extended is by the introduction of structured programming, and its related methodologies such as "top down programming" [22], "provable programming" [36], and "stepwise refinement" [16]. These methods encourage a programmer to write programs in a disciplined way so that the resulting programs will be clearer, more reliable, and more verifiable. The hierarchical nesting of segments in a plan reflects the basic notion of top down refinement. The behavior of each segment is explained solely in terms of its subsegments. However, plans are more flexible than structured programming is often interpreted to be, because, though a plan must be strictly hierarchical, the program it describes need not be. This makes it possible to describe programs which have been transformed for efficiency reasons by factoring together pieces of code which perform logically separate actions.

Another development has been the extension of the concept of a data type, to the concept of a data abstraction. This was first done in the language SIMULA-67 [12, 15], which introduced the idea of a CLASS enabling some of the data and procedural aspects of computational objects to be unified into a single concept. This concept has been extended in the languages CLU [60, 61], and ALPHARD [112, 113]. Zilles [116] and Guttag [40] have developed a formal descriptive system, called a data algebra, for describing the behavior of a data abstraction. An important aspect of data abstractions is that they can be used as a basis for modularity in a system of programs. Neither plans nor PBMs deal directly with the structure of data. The PA as a whole deals with data structures by using techniques similar to data abstractions.

The goal of automatic programming research [4] has been to create a system which will automatically generate a correct and reasonably efficient program given its specifications in some convenient form. Systems have been designed using specification through examples (such as I/O pairs [42, 89, 95], and traces [10]), exact specifications in a predicate calculus like language [65, 67], and English language descriptions [5, 84]. If an automatic programming system could perform over a wide range of applications, it would be an appealing solution to the present problems in programming. It should be noted, however, that such an automatic programming system would not make programming trivial, because developing a complete and precise description of a desired computation is difficult whether or not the description is algorithmic. In any case, it has not yet been possible to develop a wide ranging automatic programming system.

A major problem has been the fact that a lot of domain specific knowledge must be realized in the programs produced by an automatic programming system. If the user is not going to provide this information, then it must either be known to the system in advance, or reinvented by the system in response to a problem. If a system is going to operate in a wide range of domains, it is very difficult to give it all of the relevant information about each domain in advance. Unfortunately, it is even more difficult to design a system which can invent the information it needs in response to a problem. To deal with these problems, most automatic programming systems have been designed with restricted domains of applicability, so that they can have a large amount of knowledge about their domains. An example of this is PROTOSYSTEM-1 [84, 85] which has extensive knowledge of efficient file and data manipulations for business programming applications.

An interesting example of an automatic programming system is the PSI system at Stanford [37, 38]. As part of this system, Barstow [6, 39] has compiled a body of machine-usable knowledge about an area of LISP programming. The applicability of the PSI system is broadened by the requirement that the specifications for a program be expressed algorithmically in a very high level language. The knowledge of LISP programming consists of rewrite rules that progressively refine the description of the desired program into a correct implementation. The rules are annotated with pragmatic information which the PSI system uses in order to select an efficient implementation from among all the implementations possible [7, 53].

The PA system has a different emphasis from an automatic programming system. In a given situation, a typical automatic programming system either produces a complete and correct program or nothing at all. In contrast, the PA is designed around the idea of partial performance. It only tries to generate small parts of the program. Its primarily role is to understand what is going on and act as a critic while the programmer writes the program. The programmer can refer directly to algorithms and data structures known to the PA, in order to get it to generate common pieces of code. The emphasis is on cooperation between the programmer and the PA, rather than on the PA doing all the work. In order for this to be possible, the PA's internal representation for the program (namely a plan) must express the logical structure of the program in a way which is understandable to both the PA and the programmer. Plans are designed to make it easy for the PA and the programmer to reason about, and modify a program which is being developed. This is in contrast to a system like the PSI system which is based on knowledge about a program that is oriented primarily towards synthesis rather than toward program analysis or explanation. In the PSI system, the only description of the logical structure of the program being written is the history of the rule applications used to produce it. This tree structure is adequate for representing goal-subgoal relationships, but does not make a distinction between a step which just happens to precede another step and a step which satisfies a precondition required by a following step. Furthermore, the restriction to a tree structure makes it impossible to describe a program where one part of the program serves more than one purpose.

Automatic verification is another approach to the software problem. The first step towards automatic verification was the development of ways to express statements about programs as theorems to be proved. This was first done by FLoyd and Hoare [26, 45]. Pratt [78] has tightened the foundations of the Floyd-Hoare system, deriving some interesting computability results about the formalism. Manna, Waldinger [66], Pnueli [64], and Pratt [62, 78] have developed ways for talking about the dynamic behavior of a program.

A variety of automatic verification systems have been developed [50, 56, 57, 63] which verify programs written in axiomatized languages such as PASCAL [47]. They all share the same basic fault that automatic programming systems have, namely that they do not work with programs of realistic size and complexity. These systems have been extended to use heuristics [13, 54, 71, 104] and interaction with the user [21] in order to speed the verification process. A particular area of difficulty is the determination of loop invariants. This has been attacked heuristically [33, 55, 109, 110] and by analysis of special cases [8, 9, 26]. Section II.3.3 discussed how PBMs can aid in this process by breaking a loop apart. Another way automatic verification has been extended, is in the direction of dealing more directly with complex data structures and side-effects [46, 100]. This has been a particular interest of Howard Shrobe as part of the PA project [79, 91, 92].

The PA is designed to perform a logical analysis of a program, rather than to automatically verify selected properties of the program. On the one hand, it steps back from the goal of automatically verifying large programs. It does perform a large number of deductions about a program, however, it is designed so that each of these deductions is small and within the capabilities of current theorem provers. On the other hand, the PA goes beyond typical automatic verification systems, by developing a representation for the logical structure of a program (the justification links in the plan) which can then be used to assist further reasoning about the program. The plan representation is designed to be robust, in that it is useful even when it is incomplete, and in that small changes in the program usually lead to only small changes in the plan. The later property facilitates the analysis of program modifications. It is expected that the programmer will provide some of the information needed in the plan while the PA will use its deductive abilities to fill in the rest.

Another area of research which has led up to the PA is the development of various tools which can assist a programmer. Examples of this are the programming environment developed by Teitelman for INTERLISP [101, 102] and beyond [103], the programmers workbench [23], and the

MIT LISP machine environment [11]. Further examples of tools include special editors designed to work with the syntax of a particular programming language [24], and specialized error detection systems such as the DAVE system of Osterweil and Fosdick [28, 73, 74] which detects anomalous patterns of data flow in a system of programs. The PA continues the trend of combining many useful tools together in a single programming environment. A key aspect of the tools mentioned above is that, like improved programming languages, they are basically syntactic. They do not take what the programmer is doing, or why he is doing it, into account. Rather, they only try to ensure that he does what ever he is doing in a reasonable way. In contrast, the PA is designed to take the intentions of the programmer into account. This makes it possible for the PA to have additional programming aids which are more powerful than the ones above.

The earliest definitive description of a system like the PA was given by Winograd [111]. He suggested the construction of a unified programming environment including debuggers, programming language systems, and a knowledge base. Winograd also suggested the use of annotation on the program in order to help the system understand the goals and methods of the programmer. In an earlier paper, Floyd [27] also proposed a unified programming environment. However, his proposal did not include a data base for storing implementation and domain dependent knowledge. Neither of these proposals describes how to actually implement such a system.

The term "Programmer's Apprentice" was coined by Hewitt and Smith in [44] (which is based in part of the work of Smith, Lieberman, and the author [93]). Their apprentice is built around a new programming language (PLASMA) and a new model of computation ("actors"). In their approach, a system of programs is represented as a set of actors (which are essentially equivalent to subroutines). Each actor is annotated with a description of its behavior called a "contract". The main activity of their apprentice is "meta-evaluation" which symbolically executes each actor in order to show that it satisfies its contract. Yonezawa [114, 115] has developed the idea of meta-evaluation further, particularly in the direction of operating on parallel programs.

The kind of PA described in this report was originally designed by Rich and Shrobe [79]. This PA is similar to the one of Hewitt and Smith in that a system of programs is broken up into a set of pieces (segments) each of which is associated with a description of its behavior. Also like their system, the deductive component is based on symbolic evaluation. Unlike their system, the PA of Rich and Shrobe is designed to work with a standard language (LISP) rather than a new language and formalism. The most important difference between the two systems is that Rich and Shrobe incorporated the idea of a plan which directly represents the logical structure of a program. The author designed a PA [105], very similar to the PA of Rich and Shrobe, intended to work in the restricted domain of mathematical FORTRAN programs. Since that time, the development of the PA has been pursued by a group consisting of Rich, Shrobe, and the author [81, 82, 92, 107].

The kind of plan used by the PA has its origins in the work of Sussman [96] and Goldstein [34]. Both Sussman's HACKER system, and Goldstein's proposed system, MYCROFT, used a plan to represent the logical structure of the program it was working on. Since that time, the idea of a plan has developed in several directions. Sacerdoti [86] has investigated the interaction between temporal and causal constraints in plans, using a representation called "procedural nets". More recently, Genesereth [30] is using a plan representation as part of the knowledge base for an on line advisor facility for MACSYMA, a large symbolic mathematics system.

The continuing work of Sussman [97, 99] has been oriented toward the domain of electronic circuits which turns out to be similar to the domain of programming in many ways. Brown [14] explored the use of causal and teleological reasoning in the troubleshooting of complex electronic systems. A central part of his approach is the use of a plan for a circuit. DeKleer [17, 18] is developing a deeper theory of plans for electronic circuits. One of the central features of a plan is the existence of explicit logical links which show the logical structure of a program or a circuit. The creation and use of these links to control the reasoning process has been investigated by Sussman [94, 97] and Doyle [19, 25]. Two prime uses of the links are in the control of backtracking and as a means for saving a summary of a deduction in a convenient form for later use. The PA is designed to use these links in similar ways.

Starting from the original work of Goldstein, Miller and Goldstein [35, 68, 69] have moved in the direction of developing a psychologically plausible theory of the problem solving process of programming in highly constrained domains such as the programming of a computer controlled cursor to draw stick figures on a TV screen. They have made a catalog of very general problem solving strategies, such as decomposition and reformulation, and have organized them into an augmented transition network grammar which can be used to generate and recognize simple drawing programs. their current notion of a plan is quite different from the one presented here because it is oriented toward explaining the problem solving process rather than explaining the structure of a given program.

The basic notion of a plan described here was originally introduced by Charles Rich and Howard Shrobe [79]. The author described in [105] a notion of a plan based on their notion introducing the idea of a PBM and the idea of viewing a loop as a composition of stereotyped loop fragments.

Another important area of research which is a precursor of the PA is the compilation of libraries of known algorithms, transformations, and methods of combining operations together. Much of this work has been directed toward easing the programming task and is not involved with any automatic system. For example, Knuth [59] has compiled a compendium of fundamental algorithms. Gerhart [31, 32] has compiled a set of preproven algorithms and correctness preserving program transformations. In [87] Shwartz proposes the construction of a library of proved root programs which represent the "fundamental and essentially undecomposable elements of algorithmic technique".

Automatic programming research has led to the compilation of programming knowledge in a form which can be used by automatic systems. For example, PROTOSYSTEM-1 uses a catalog [84, 85] of algorithms and other knowledge about business data processing programs. Similarly, The PSI system uses a knowledge base [6, 39] of information about an area of LISP programming. The PA will have a library of commonly used data structures and plans for common algorithms. As part of the PA project, Charles Rich [79, 80, 82] is particularly interested in how this knowledge should be represented, what information should be included, and how the PA can recognize instances of data structures and algorithms it knows about in a program. An early attempt in this direction was the work of Ruth [83]. He constructed a system which analyzed correct and near-correct PL/I programs from an introductory programming course, giving specific comments about the nature of the errors detected in the incorrect programs. In Ruth's system, the class of expected programs is represented as a formal grammar augmented with global switches that control conditional expansions. This grammar is then used in a combination of top-down, bottom-up and heuristic parsing in order to recognize particular programs. Ruth's analysis procedure would not be sufficient for the PA due to the fact that the result of his analysis does not contain specifications or logical links describing the logical structure of the program. In addition, the PA has to be designed in a more flexible way because it is not looking for instances of one particular class of programs, but rather trying to analyze arbitrary programs built up out of familiar components.

The PBMs are a special case of a catalog of programming knowledge. They are more general than the kind of knowledge discussed above, being more akin to programming language constructs. As discussed above, the straight-line PBMs are very similar to structured programming language constructs. In addition, though the recursive PBMs do not appear in full form in programming languages, they embody natural ideas and have features in common with programming language constructs such as DO, the MAP functions in LISP[70], GENERATORS in ALPHARD [90], ITERATORS in CLU [61], and some of the vector operators in APL [76]. The language most closely related to the recursive PBMs is the one proposed by Kahn, and MacQueen [51, 52]. Their language makes it possible to construct coroutine processes which operate directly on temporal sequences of values. An initial outline of the PBMs was presented by the author in [105]. This included the basic idea that loops can be analyzed as consisting of a combination of loosely coupled loop fragments. Since that time, the PBMs have been developed to their current state [107]. One of the important changes has been the maturation of the recursive PBMs in order to make the analogy with composition complete. In order to make this analogy precise, the notion of a temporal sequence of values was developed jointly by the author and Howard Shrobe.

A recent article by T. Pratt [77] calls for an improvement in the design of loop control structures. He focuses on the "control computation" in a loop. This is the computation which controls how many times a loop will be executed and when it will be exited, as distinct from the computation in the loop which performs actions needed by the program as a whole but which does not affect the termination of the loop. His basic observation is that a small number of common control computations appear in a large percentage of loops which are written, and that unfortunately, the common looping constructs obscure rather than highlight this fact. He presents several criteria for judging improved loop constructs. His principle criteria are that the parts of the control computation including any initialization should be grouped together, rather than spread through the loop, and that stereotyped control computations should be clearly identified as such. Among other things, the recursive PBMs are designed to meet exactly these two goals with regard to all of the computation in a loop, not just the control computation. However, it should be noted that the PBMs are not exactly what Pratt is calling for, because they are designed to be used by an automatic system while he is talking about ways to improve programming language constructs. As mentioned above, it is not clear that it is easy to produce structured programming language constructs based on the recursive PBMs.

A program can be analyzed in terms of PBMs in order to construct a PBM plan which can then be used to aid in further reasoning about the program. The process of PBM analysis can be compared with other ways of analyzing a program. Analysis such as that of Rich [82] and Ruth [83] which looks for particular algorithms in a program is much more specific than PBM analysis. PBM analysis uses more general criteria in order to break up a program into loosely coupled pieces. It is hoped that an initial analysis in terms of PBMs will facilitate a later analysis in terms of specific algorithms.

Another type of analysis which has been extensively studied is analysis of a program based on its "flow graph". A flow graph is a directed graph where the nodes correspond to statements in the program, and the arcs correspond to the control flow between the statements. A flow graph is very similar to a flow chart, or to a surface plan where the data flow links have been removed. Paige [75] surveys several ways in which a flow graph (and therefore the corresponding program) can be analyzed in order to break it up into sections. One way of breaking up a flow graph is based on the idea, introduced by Allen and Cocke [2], of an interval. This calls for locating single entry loops in the flow graph. It is essentially equivalent to locating instances of the PBM single self recursion. Baker [3] has developed a system based on intervals which in effect is capable of analyzing a program in terms of the PBMs composition, conditional, and single self recursion and then printing out a structured version of the program. A different way to analyze a flow graph is based on the idea of a "class". In this approach, the flow graph is represented by a regular expression. The regular expression is then broken up into subexpressions. Each subexpression corresponding to a class. The prime difficulty in this approach is the selection of a convenient regular expression to represent the flow graph.

A fundamental difference between PBM analysis and any analysis based on flow graphs comes from the fact that flow graphs do not contain arcs corresponding to data flow. As a result, all the segmentation procedures for flow graphs are based on control flow. In contrast, PBM analysis is applied to a surface plan which does contain data flow arcs. Data flow arcs are the primary basis for the parts of PBM analysis involved with locating initializations and analyzing loops in terms of the PBMs temporal composition, augmentation, filter, and termination. This part of PBM analysis breaks up a program in a fundamentally different way from any analysis based on a flow graph.

III. The Structure of a Plan

The notion of a plan is still evolving. The description below is a snapshot of its current state. The description is divided into three parts: a description of how specifications are specified, a description of how the interaction of segments is specified, and a description of how the logical structure of a program is specified.

III.1 Describing What a Segment Does

The basic unit of a plan is called a "segment". A segment is a logical entity analogous to a subroutine which corresponds to a part of a program. It usually corresponds to a contiguous section of the code for the program though it does not have to. For example, the lines "Z=0" and "Z=Z+A(I)" in Figure II-6 can be grouped together into a single logical segment as in Figure II-7. As part of the plan for a particular program, each segment is linked by an "implementation link" to the part of the code for the program corresponding to it. This will be discussed further in the next section. The first level of description of a segment specifies what its behavior is, apart from how this behavior is achieved. For the purpose of this description, the segment is treated like a black box and given an input/output specification called a "behavioral description".

The behavioral description for a segment consists of several parts. A segment has a set of "input" ports. These ports indicate how many inputs the segment has and provide names for the input objects which can be used in logical expressions which refer to them. Similarly, a segment has a set of "output" ports. The existence of an input port implies that the segment always requires that input in order to operate. The existence of an output port similarly implies that the segment always produces that output. As part of the plan for a particular program, each port is linked via an implementation link to the part of the code for the program corresponding to it. These implementation links are not shown in the examples below. Instead, the names of the ports are chosen mnemonically in order to indicate the correspondence. The most common device will be to give a port the same name as a variable which carries data to or from the port. However, it should be noted that there is no special relationship between variables and ports. In a plan, the name of a port actually has no intrinsic meaning. Its only use is as a unique identifier for the port. The only requirement is that no two ports in the same case of the same segment can have the same name.

A segment has a set of "pre-conditions". Each pre-condition is a logical expression which can refer only to the inputs of the segment and global information. The meaning of a pre-condition is that the segment's behavioral description is guaranteed to be accurate only in situations where the pre-conditions are satisfied. A segment also has a set of "post-conditions". These are logical expressions which can refer only to inputs and outputs of the segment, and global information. The behavioral description states that if the pre-conditions are true immediately before the segment is executed, then the post-conditions will be true immediately after it is executed if it terminates. This is a familiar idea which is expressed by Hoare [45] as "pre-conditions {segment} post-conditions". The language currently used for the pre-conditions and post-conditions is a kind of predicate calculus (its exact nature is not important in the context of this report). In general, it is assumed that in the background underlying a plan there is a data base containing general information which can be referred to. For example, it would contain facts about data structures, algorithms, and mathematics.

Figure III-1 shows an example of a behavioral description. The behavioral description forms four of the parts of a plan. The line beginning "inputs:" lists the inputs of the program. The line beginning "pre-conditions:" lists the logical expressions which are the pre-conditions of the program's behavioral description. The line beginning "outputs:" lists the outputs of the program. The line beginning "post-conditions:" lists the logical expressions which are the post-conditions of the program's behavioral description.

```
Z = 0;
DO I=1 TO N;
   IF A(I)>0
      THEN Z = Z+A(I);
END;
```

```
          Segment A (the program above)
           inputs: A, N
   pre-conditions: REAL(A), VECTOR(A), INTEGER(N), 0≤N≤SIZE(A)
          outputs: Z
  post-conditions: REAL(Z), Z=Σ{i| 1≤i≤N ∧ A(i)>0}A(i)
```

Figure III-1: A behavioral description.

The behavioral description in the figure describes the program as taking two inputs: A and N. The names of the input ports have been chosen to be the same as the names of the variables which bring the corresponding values into the program in order to indicate the relationship between the behavioral description and the program. One input is a vector of real numbers, and the other is a non-negative integer which does not exceed the bounds of the vector. The plan indicates that the program has one output: Z. This output is a real number and equals the sum of the positive members of the first N elements of A. The pre-conditions and post-conditions are listed separated by commas. Logically, each list corresponds to a conjunction of the items in the list.

Sometimes the easiest way to describe a segment is to say that it acts in one of a set of ways depending on the circumstances. The notion of "cases" is introduced into a behavioral description in order to allow this kind of description. Given a behavioral description with n cases, each input, output, pre-condition, and post-condition is indexed according to which case it refers to. Looking at the behavioral description on a case by case basis, this provides n complete descriptions of what the segment does. The behavioral description specifies that a given instance of executing the segment will behave exactly like the description associated with one of the cases. When considering only this single execution, all of the information associated with the other cases can be ignored. To this end, a pre-condition or post-condition in one case can only refer to inputs and outputs in the same case. If an input, output, pre-condition, or post-condition applies equally to every case, then it is said to be in the "NIL" case and is not indexed as applying to any one case. If a behavioral description has only one case (as in Figure III-1) then it is usually represented as having no explicitly named cases. Rather everything is put in the NIL case.

The final part of a behavioral description is a "condition". A condition is a logical expression referring only to inputs of the case it is associated with and global information. The set of conditions attached to a case specifies the situations in which the description associated with that case is the correct description for the behavior of the segment. Taken as a group, the sets of conditions associated with the n cases of a segment must be mutually exclusive so that there will never be any ambiguity about which case is applicable. Further, it is an implied pre-condition of the behavioral description as a whole that the conditions as a group cover all possible situations in which the segment can be executed so that there will always be a case which is applicable. Together, the pre-conditions, post-conditions, and conditions for a segment are referred to as its specifications.

Figure III-2 shows a behavioral description which uses cases. It introduces two new parts of the plan representation. Labels of the form "CASEn" are used to identify the parts of a plan which only apply to one case. The parts of the plan which precede the first case label are in the NIL case and apply to all cases. The number in the case label serves as a name for the case. The lines beginning "conditions:" give the logical expressions which are the conditions of a case.

```
                 IF A(I)>0
                    THEN Z = Z+A(I);

          Segment A (the if-then-else above) (not using cases)
           inputs: A, I, Z
   pre-conditions: VECTOR(A), REAL(A), INTEGER(I), REAL(Z), 1≤I≤SIZE(A)
          outputs: ZOUT
  post-conditions: REAL(ZOUT), ZOUT=(IF A(I)>0 THEN Z+A(I) IF A(I)≤0 THEN Z)

          Segment A (the if-then-else above) (using cases)
           inputs: A, I
   pre-conditions: VECTOR(A), REAL(A), INTEGER(I), 1≤I≤SIZE(A)
        CASE1
       conditions: A(I)>0
           inputs: Z
   pre-conditions: REAL(Z)
          outputs: ZOUT
  post-conditions: REAL(ZOUT), ZOUT=Z+A(I)
        CASE2
       conditions: A(I)≤0
```

Figure III-2: A behavioral description using cases.

The figure shows two behavioral descriptions for the same program. This highlights the fact that there are many ways a given piece of code can be described. Which one is best depends on how the description is going to be used. In a plan it is quite possible for several different behavioral descriptions of segments to coexist which all refer to the same piece of code.

The first behavioral description in the figure does not use cases. It states that if A(I)>0 then A(I) is added to Z otherwise, it is not. One possible defect in this behavioral description is that it specifies that even when A(I)≤0, the segment requires the input Z and produces the output ZOUT.

The second behavioral description describes the program differently. It specifies that the program always has the two inputs A and I. However, the rest of the behavioral description has two cases. When A(I)>0 then the program takes an additional input Z and produces the output ZOUT. In the other case, the segment does not do anything. Cases must be used in order to describe a segment as having optional inputs or outputs. This is due to the fact that the appearance of an input or output in a case implies that it is obligatory in that case. It will be seen below that there are other situations which require the use of cases.

III.2 Describing How a Segment Does What It Does

The last section described the parts of a plan which describe what a segment does. This section describes the parts of a plan which describe how a segment does what it does. A segment is described as containing a set of subsegments which interact by means of control flow and data flow in order to create the behavior of the outer segment. Together, the subsegments, data flow, and control flow specify an algorithm which can be used to achieve the behavior of the containing segment. In this algorithm, the subsegments are looked at as black boxes. Only their behavioral descriptions are relevant, not how they, in turn, achieve their behaviors.

In a plan the subsegment relation is represented by a subsegment arc from one segment to another. Logically, a segment can only be a subsegment of one other segment, and cannot be a subsegment of itself either directly or indirectly. As a result the subsegment arcs in a plan form a tree with a segment at each node. This is not affected by the fact that several different segments can refer to the same part of the code for a program. The segment in a plan which is not a subsegment of any other segment is called the outermost segment, and is the root of the tree of subsegment arcs. A plan is usually referred to as a plan for its outermost segment. A segment in a plan which has no subsegments in the plan is referred to as a terminal segment. The other segments in a plan have both subsegments and supersegments, and are referred to as intermediate level segments. In a plan, terminal segments usually correspond to applications of primitive functions such as +, -, and >. A plan contains no information about the internal structure of a terminal segment. One thing which should be noted here involves the type token distinction. A primitive function such as + corresponds to a type. A particular application of it is a token of that type. Every terminal segment in a plan is a token. Every non-terminal segment is a type, defined by the interaction of its subsegments. Terminal segments in one plan can by tokens which are instances of types defined in other plans.

As mentioned above, the plan for a particular program contains implementation links which show what parts of the code for the program correspond to each segment. Each terminal segment has an implementation link going from the segment to a construct in the program, such as an instance of a function application. A non-terminal segment can have an implementation link going to a function definition or some intermediate level construct in the program. A non-terminal segment can also have no implementation link in which case the segment corresponds to the set of things which correspond to its subsegments. Segment implementation links are not shown in the examples below. Rather the names of the segments are chosen mnemonically in order to indicate what the implementation links are. For example, a terminal segment which corresponds to an application of a primitive function (such as "*") will be given a name (such as "*", "*1" or "*A") derived from the name of that function. Giving the terminal segment the name "*" does not indicate that the segment is the type "*". As mentioned above, terminal segments are tokens while non-terminal ones are types. A segment which corresponds to the definition of a function will usually be given the same name as the function being defined. Note that like the name of a port, the name of a segment has no intrinsic significance in a plan. The only requirement is that no two segments in the same plan have the same name.

Subsegments are connected with each other and with their supersegment by two kinds of arcs: control flow arcs and data flow arcs. The form of plan representation described here requires that there be no loops in control flow or data flow. Without loss of generality, loops in a program are expressed in a plan by means of recursion. The primary affect of this restriction is that it implies that when a segment is executed, each of its subsegments will be executed at most once. This simplifies the definitions of control flow and data flow given below.

III.2.1 Control Flow Arcs

Control flow arcs connect the output sides of segments or cases of segments with the input sides of segments or cases of segments. They indicate the flow of control from a case of a segment to a case of another segment. In a program, control flow can be implemented by a variety of mechanisms such as GOTOs, sequential placement of statements, and nesting of expressions. Control flow links abstract away from these mechanisms and represent control flow information directly and explicitly.

Three basic types of control flow can be identified. A control flow can go from the output side of a case of a subsegment to the input side of a case of a subsegment. This indicates that the first subsegment's case must terminate execution before the second subsegment's case can start execution. A control flow can go from the input side of a case of a supersegment to the input side of a case of one of its subsegments. This indicates that the case of the subsegment cannot start execution until the execution of the indicated case of its supersegment begins. Analogously, a control flow can go from the output side of a case of a subsegment to the output side of a case of its supersegment. This indicates that the execution of the case of the supersegment cannot terminate until the execution of the indicated case of its subsegment terminates. A control flow from the input side of a supersegment to the output side of the supersegment only implies that the termination of the segment must follow the beginning of the segment's execution and is a degenerate case. Control flows cannot hop over a segment boundary. They can only connect a segment with one of its subsegments, or connect two subsegments of the same segment.

The way control flows restrict the order of execution can be summarized as follows. The input side of a case of a segment is said to be "activated" when that case of the segment begins to be executed. Similarly, the output side of a case of a segment is said to be "activated" when that case of the segment completes execution. It is clear that the output side of a case of a segment cannot be activated until after the input side of that case of the segment is activated. Using the above definition of activate, the action of a control flow can be completely specified by saying that it signifies that its destination cannot be activated until after its source is activated. The subsegment relation also constrains activation as follows. No case of a subsegment can be activated unless the input side of a case of its supersegment is activated. This is equivalent to saying that each subsegment arc acts, among other things, like a control flow arc. Note that plans do not make the assumption that execution will be serial. They permit segments to be evaluated in parallel where possible. There is a background assumption of eventual fairness. It is assumed that every segment which becomes eligible for execution will eventually be executed.

In general, a plan is not required to have control flows. An absence of control flows merely implies an absence of execution order constraints. This is used in a plan to represent potential parallelism. In the examples below, control flows will usually be omitted wherever they are not necessary.

Figure III-3 shows how control flow and the subsegment relation are represented in a plan. The program in the figure is a simple loop. In order to represent it in a plan, it is first converted to an equivalent recursive program as shown. Any loop can be converted into a singly recursive program which is "tail recursive" [98]. A tail recursive program is a recursive program where the recursive call is the last action taken by the program. This means that when the recursive call returns, the program immediately returns without performing any additional computation. The conversion to recursive form is done so that there will not be any loops in data flow or control flow in the plan. The figure shows only the subsegments and control flow in the plan. Figure III-4 shows the behavioral descriptions of the segments and the data flow in the plan.

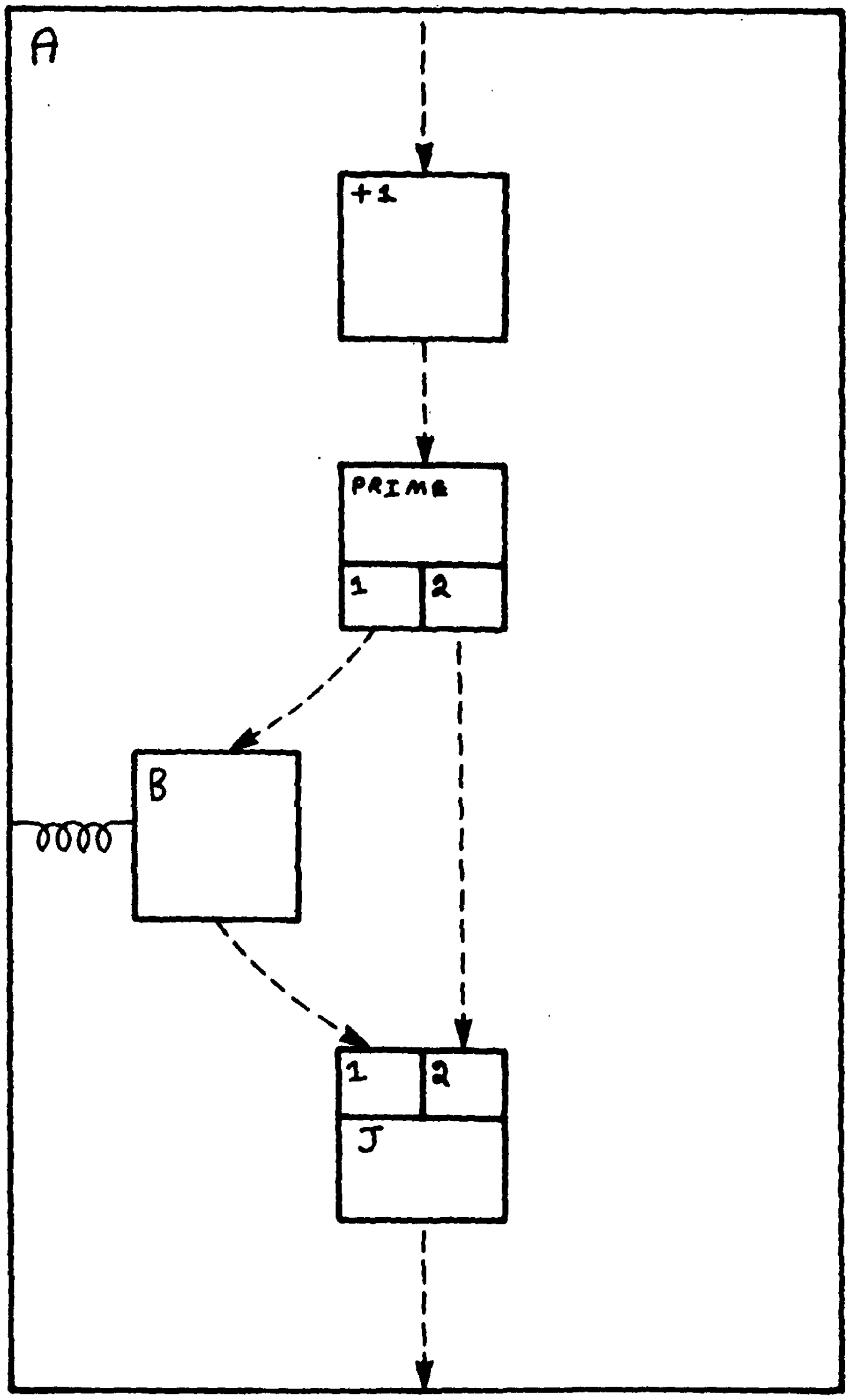
The line beginning "subsegments:" represents each subsegment arc which originates on the segment by listing the names of the subsegments of the segment. In the example the supersegment A corresponds to the entire program. The subsegment +1 corresponds to the operation of adding one to K. It is given the name "+1" in order to indicate this correspondence. The subsegment PRIME corresponds to the function call which determines whether K is prime. Note that in this plan, PRIME is a terminal segment. Another plan might well specify internal structure corresponding to the function being called, defining the type of which this terminal segment is a token. The subsegment B corresponds to the recursive invocation of the loop. Subsegment J corresponds to the joining of control flow at the exit from the loop.

The line beginning "recursive link:" in the plan for segment B indicates that segment B is an instance of segment A. A recursive link between two segments indicates that they have exactly the same plan, the same behavioral description and the same internal structure. The recursive link is used to represent the fact that the plan in the figure is actually infinite in extent. Recursive links appear to block substitutability of segments because they require that the internal plans for two segments be identical, which seems to preclude substituting for just one of them. However, recursive links are looked at as purely an abbreviation in a plan. If a plan is unabbreviated, it may become infinite but that will not block substitutability. If the plan is reabbreviated after a substitution is made, then the resulting recursive links may be different which indicates that substitution can indeed affect recursive links.

The line beginning "control flow:" lists the control flow arcs in the plan. Each arc is represented as "source→destination", where both the source and the destination are built up by concatenating the name of a segment with the case name (if it is not NIL). Note that in the representation for a control flow arc, it is not necessary to specify the sides of the cases involved. This is because this information is redundant with the subsegment relation. If a control flow goes from segment X to segment Y (i.e. X→Y), then if Y is a subsegment of X the control flow must go from the input side of X to the input side of Y; If X is a subsegment of Y the control flow must go from the output side of X to the output side of Y; otherwise, the control flow must go from the output side of X to the input side of Y. For example, A→+1 means that there is a control flow from the input side of segment A to the input side of segment +1. Similarly, PRIME1→B means that there is a control flow from the output side of case 1 of segment PRIME to the input side of segment B.

The figure also shows a "flow diagram". A flow diagram is a graphic representation of the structural information in a plan. It does not represent any information which is not in the plan. It is used in order to help visualize the information in the plan. In a flow diagram, segments are represented by boxes. The subsegment relation is indicated by nesting of these boxes. The name of each segment is written in the corresponding box. The recursive link between segments A and B is represented in the flow diagram by a looping line linking the two segments. Control flow arcs are represented by dashed lines. An arrow head indicates the destination end of each arc. The rectangle at the top of a segment box represents the input side of its cases. If there is more than one case, then the rectangle is subdivided into sections which are given numbers corresponding to the case names. The rectangle at the bottom of a segment box represents the output sides of its cases in the same way. Strictly speaking, if the input side is subdivided then the output side should also be subdivided and vice versa. This is not the case in the figure. However, this is just an abbreviation. The non-subdivided rectangles correspond to the NIL case and merely indicate that there are no interesting distinctions between the cases on that side. (As in the figure, non-subdivided rectangles are usually omitted.)

```
          LOOP: K = K+1;
                IF ¬PRIME(K) GOTO LOOP;

                    An equivalent recursive program
                PROCEDURE NEXT-PRIME(K);
                   IF ¬PRIME(K) THEN RETURN(NEXT-PRIME(K+1));
                                ELSE RETURN(K);
                END;

          Segment A (the whole program above)
     subsegments: +1, PRIME, B, J
    control flow: A→+1, +1→PRIME, PRIME1→B, B→J1, PRIME2→J2, J→A

          Segment +1

          Segment PRIME
       CASE1
       CASE2

          Segment B
  recursive link: A

          Segment J
       CASE1
       CASE2
```



Flow Diagram for Figure III-3

Figure III-3: An example of control flow.

The program in the figure is a simple loop which finds the first prime larger than the initial value of K. Consider what the control flows signify in the plan. The control flow A→+1 indicates that when the input side of segment A is activated then the input side of segment +1 will be activated. Similarly, the control flow +1→PRIME indicates that the subsegment PRIME is activated after +1 finishes execution. When PRIME is executed, either its first case or its second case will be executed depending on its conditions. They are mutually exclusive, so exactly one of the output sides of PRIME will be activated. Suppose case 1 is executed. Control flow PRIME1→B will cause the activation of segment B (which is a recursive instance of the outer segment A). After this terminates, control flow B→J1 will activate case 1 of segment J. After this case executes, control flow J→A allows the outer segment A to terminate execution. Alternatively, case 2 of PRIME could be activated in which case control would pass directly through J to the output side of A bypassing B. This corresponds to an exit from the loop because it causes all of the invocations of A to terminate one by one.

Three types of segments can be identified based on how they interact with control flow. A segment, such as PRIME in the figure, which has more than one case, and where the control flows leaving the output sides of two different cases, go to different places is called a "split". A segment like J which has more than one case, and where the control flows terminating on the input sides of two different cases come from different places is called a "join". Splits and joins are the other situation besides optional inputs and outputs where a behavioral description is forced to have cases. Segments like A, +1, and B which are neither a split nor a join are called "straight-line". Segments which are both splits and joins are not allowed. In the current design of plans, the only joins which are allowed are terminal segments like J which have no subsegments.

The main purpose of a split is to perform a test and to encode the results of this test as a choice between alternate control flow paths. Control flow enters a split through its NIL case. The conditions determine which case will be executed, and hence which output control flow will be taken. If a split is a non-terminal segment, then its internal structure must be such that it determines which output case will be activated.

The purpose of a join is to join separate control flow paths together. Note that there are no explicit conditions associated with the cases of a join. Which case is executed depends on which one is activated by its incoming control flow. The incoming control flows act as implicit conditions, as do incoming data flows (see below). As conditions, they must be mutually exclusive. There must be only one control flow active in any given situation.

Looking at control flows a little deeper, consider what "fan-out" of control flows implies. If two or more control flows originate on the same side of the same case of the same segment, then they are said to fan-out. An example of this is the control flow +1→PRIME in the figure. This control flow goes from the NIL and only case of +1 to the NIL and hence both cases of PRIME. It is really an abbreviation for the two control flows +1→PRIME1 and +1→PRIME2. These control flows specify that neither case of PRIME can be activated until after +1 terminates execution. The meaning of a group of fanning-out control flows is simply the conjunction of the meanings of the individual control flows.

The two control flows originating on PRIME are not a case of fan-out because they originate on different cases. They have a different meaning then if they were fanning-out. They do not just specify that neither B nor case 2 of J can be executed until after PRIME is executed. Rather, they specify that B cannot be executed until after case 1 of PRIME is executed, and case 2 of J cannot be executed until after case 2 of PRIME is executed. The difference between these two specifications is part of what makes a split a split.

"Fan-in" of control flow is analogous to fan-out. If two or more control flows terminate on the same side of the same case of the same segment, then they are said to fan-in. This implies that the destination cannot be activated until after all of the sources have been activated. Note that the two control flows terminating on the join J are not a case of fan-in. They do not specify that J cannot be executed until after both case 2 of PRIME terminates and B terminates. In fact, notice that it is impossible for both case 2 of PRIME and B to terminate, because B cannot be executed unless case 1 of PRIME is executed. Rather, the control flows terminating on J specify that case 1 of J cannot be executed until after B terminates, and that case 2 of J cannot be executed until after case 2 of PRIME terminates. They specify that J cannot be executed until after either B terminates or case 2 of PRIME terminates. The difference between these two specifications is the essence of what makes a join a join.

There is an added complexity associated with fan-in of control flow. Consider the control flow J→A in the figure. This goes from the NIL and hence both cases of J to the NIL and only case of A. It is therefore an abbreviation for two control flows: J1→A and J2→A, and a case of fan-in. It would not make any sense to say that this control flow meant that A could not terminate until after both case 1 and case 2 of J have terminated, because, it is impossible for two cases of the same segment to both terminate. Rather, this control flow is taken to mean that A can terminate if either J1 or J2 terminates. In general, if two or more control flows fan-in from cases of the same segment, then the meaning of the group of control flows is taken to be that the destination can be activated if any of the sources is activated. This is called "disjunctive" fan-in in order to distinguish it from normal fan-in which is called "conjunctive" fan-in. This distinction is not needed for fan-out. All fan-out is conjunctive.

It should be noted that in most of the examples below, the only fan-in or fan-out which occurs is that which involves the NIL case as in the example above. As a result, there never appears to be any fan-in or fan-out in the diagrams due to the way these control flows are abbreviated. More complex fan-in and fan-out would occur in programs which have explicit parallelism.

III.2.2 Data Flow Arcs

Data flow arcs connect input and output ports with each other. A data flow indicates that a data item passes unchanged from one port to the other. A data flow restricts the order of execution in the same way that a control flow does. The case which holds the source of a data flow must be activated before the case which holds its destination can be activated. In fact data flow arcs can be looked at as being control flow arcs with the added property that they specify the transfer of a data object. Alternatively, control flow arcs can be looked at as being degenerate data flow arcs. One consequence of the close relationship between control flows and data flows is that if there is a data flow from a port in a case of a segment A to a port in a case of another segment B, then any control flow from the case of A to the case of B is redundant. Such control flows are usually omitted from a plan in the interest of brevity.

In a program, data flow can be implemented by a variety of mechanisms such as variables, assignment, nesting of expressions, function arguments, and side-effects on data structures. Data flow links abstract away from these mechanisms and are the only way of indicating data flow in a plan. Further, they are unambiguous. There are no side-effects in a plan which can affect the data flow. Rather the data flow arcs summarize the net effect of any side-effects which occur in a program. If a program uses side-effects, it can be difficult to determine what the data flow arcs should be.

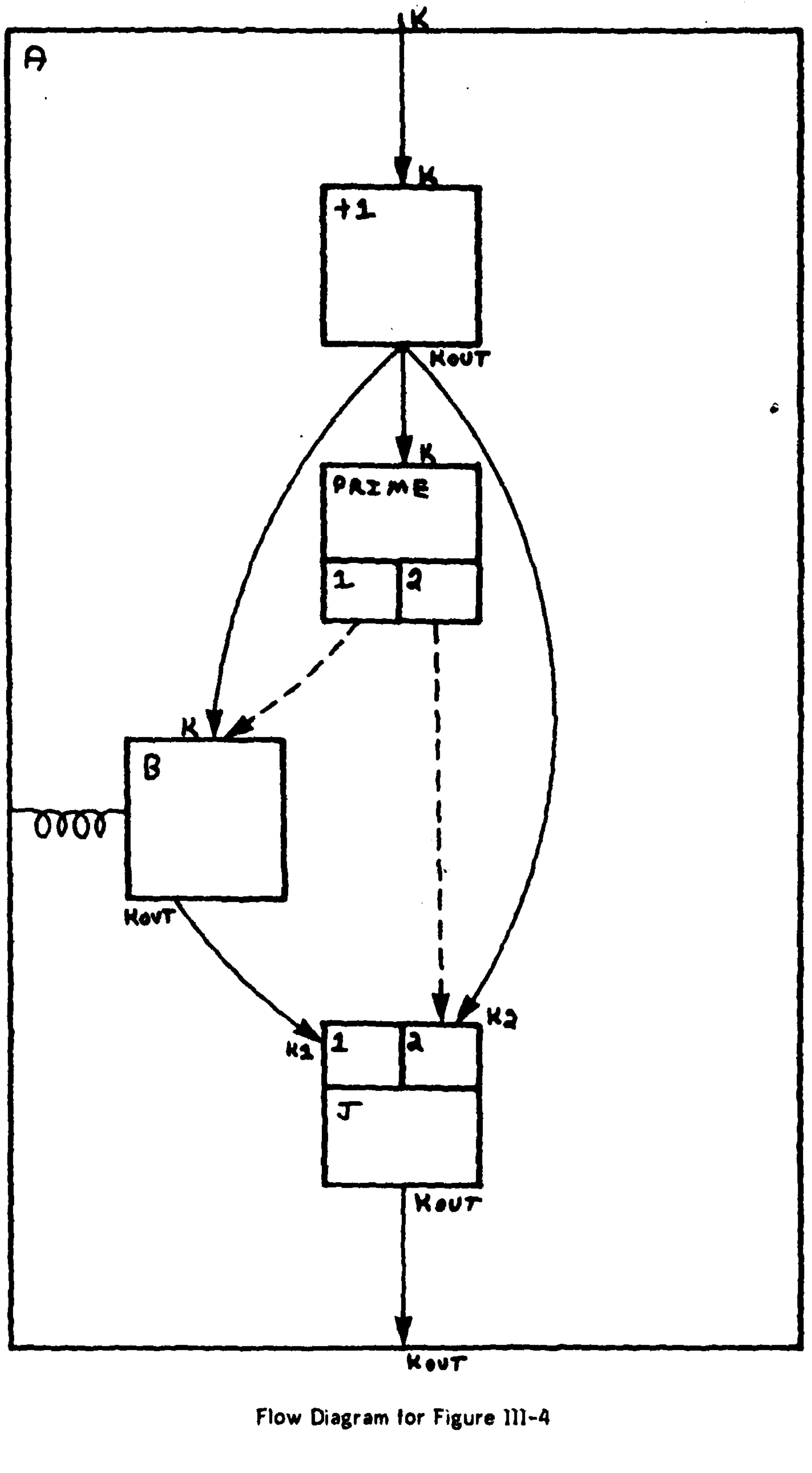
Four types of data flow can be distinguished. A data flow can go from the output of a subsegment to the input of another subsegment. This corresponds to composition of the two subsegments. A data flow can go from an input of a supersegment to an input of one of its subsegments. This indicates that the input of the supersegment is being passed down to become an input of the subsegment. Similarly, a data flow can go from an output of a subsegment to an output of its supersegment to indicate that the output of the subsegment becomes an output of the supersegment. Finally, a data flow can go directly from an input of a supersegment to an output of the supersegment. This indicates that one of the supersegment's inputs is available unchanged as an output. Like control flows, data flows are not allowed to hop over segment boundaries.

The existence of an input port on a segment implies that a value must be supplied in order for the segment to be executed. As a result, there must be a data flow arc terminating on the input because this is the only way a value can be supplied to the input. The only exception to this is the outermost segment in a plan. There is no external environment which can supply values for its inputs. Similarly, the existence of an output port on a segment implies that a value must be supplied at the port when the segment completes execution. In order for this to be true, there must be a data flow which terminates on the output port. The only exception to this is terminal segments, which have no subsegments inside them which could provide values. It is not required that there must be a data flow starting at each port. This is because the existence of a port does not imply that the value at the port must actually get used anywhere.

Consider Figure III-4. It shows the same information as in Figure III-3 with behavioral descriptions and data flow added. Data flow is represented on the line beginning "data flow:". Each data flow arc is represented by an entry of the form "source→destination", where both the source and the destination are a qualified name which consists of a segment name, possibly suffixed by a case identifier, concatenated with a period concatenated with the name of an input or output of that segment. For example, B.KOUT→J1.K means that there is a data flow arc from the output KOUT of segment B to the input K in case 1 of segment J. (There is never an ambiguity between inputs and outputs because input and output names are never the same.)

In the flow diagram in the figure, data flow is represented by solid lines with an arrow head indicating the destination of the data flow. The input and output ports of the segments are represented by the points where the data flow arcs intersect the segment boundaries. The name of each port is written next to the point representing it.

```
          LOOP: K = K+1;
                IF ¬PRIME(K) GOTO LOOP;

          Segment A (the whole program above)
           inputs: K
  pre-conditions: INTEGER(K)
         outputs: KOUT
 post-conditions: INTEGER(KOUT), PRIME(KOUT), ∀J(K<J<KOUT → ¬PRIME(J))
     subsegments: +1, PRIME, B, J
       data flow: A.K→+1.K, +1.KOUT→PRIME.K, +1.KOUT→B.K,
                  +1.KOUT→J2.K2, B.KOUT→J1.K1, J.KOUT→A.KOUT
    control flow: PRIME1→B, B→J1, PRIME2→J2
```



Flow Diagram for Figure III-4

```
          Segment +1
           inputs: K
   pre-conditions: INTEGER(K)
          outputs: KOUT
  post-conditions: INTEGER(KOUT), KOUT=K+1

          Segment PRIME
           inputs: K
   pre-conditions: INTEGER(K)
      CASE1
      conditions: ¬PRIME(K)
      CASE2
      conditions: PRIME(K)

          Segment B (which has the same plan as segment A)
   recursive link: A
           inputs: K
   pre-conditions: INTEGER(K)
          outputs: KOUT
  post-conditions: INTEGER(KOUT), PRIME(KOUT), ∀J(K<J<KOUT → ¬PRIME(J))

          Segment J
          outputs: KOUT
        CASE1
           inputs: K1
  post-conditions: KOUT=K1
        CASE2
           inputs: K2
  post-conditions: KOUT=K2
```

Figure III-4: An example of a plan with data flow.

The only differences between this figure and the last is that data flows and behavioral descriptions have been added, and that some of the control flows have been removed because they are redundant with the data flow. Consider what the data flows signify in the figure. Segment A has one input. Data flow A.K→+1.K shows that it is used by Segment +1. It also shows that segment +1 can be executed as soon as segment A begins execution. Segment +1 has an output which is used in three places as the three data flows leaving it show. Data flow +1.KOUT→PRIME.K shows that the output of +1 is tested by PRIME and that PRIME cannot execute until +1 has finished execution. The other two data flows show that the output is used by segment B and the join J. The two data paths into join J show that the output of J comes either from B or +1 depending on the situation. The data flow J.KOUT→A.KOUT shows that this value then becomes the output of A. Part of the definition of a join (such as segment J in the figure) is that it does not perform any computation. It just maps its inputs to its outputs. The statement of equality in the post-conditions for segment J says that the output value is the same as the input value. As a result, everything which is true of the input value is also true of the output value.

Fan-out of data flow from a port is common. It restricts the execution order in exactly the same way as a fan-out of control flows would. In addition, it specifies that the same data value is being used in several places. Conjunctive data flow fan-in is not allowed, because it is not clear what it would mean from the point of view of data values. If two data flows conjunctively fanned-in into a port, then that would imply that two data values would be simultaneously available. The clear implication is that they should both be used, but this contradicts the idea that the port expects a single value. Disjunctive fan-in from a join into a port is allowed. An example of this is the data flow J.KOUT→A.KOUT which is really an abbreviation for J1.KOUT→A.KOUT and J2.KOUT→A.KOUT. The execution order is restricted in the same way that it would be with disjunctive control flow fan-in. In addition, the data flows specify that the value appearing at A.KOUT is one of two different values depending on the situation.

The interaction of the join J with the data flow is very important. The control flow entering the join specifies the situations in which its two cases are executed. Namely case 2 is executed when case 2 of PRIME is, and case 1 is executed otherwise. The data flows entering the cases identify what values will be received in these two situations. The key feature of the join is that it links these two things together so that it is clear that the output of A comes directly from +1 when case 2 of PRIME is executed, and from B otherwise.

The use of joins can be contrasted with the mechanisms which were used in the plans described in [79, 105]. These plans did not use joins. Rather, they essentially specified that all fan-in of control flow and data flow was disjunctive. A join free plan for the program above would be the same as the one in the figure except that there would be no join J, and all of the data flow and control flow which terminates on J would terminate directly on the output side of A. The join free plan specifies that A could terminate after either B or case 2 of PRIME terminated, and that the value A.KOUT would come from either +1.KOUT or B.KOUT. The major problem with the join free representation is that it is not always possible to determine whether A.KOUT is +1.KOUT or B.KOUT.

For example, suppose that case 1 of PRIME is executed and then B. The control flow then allows A to terminate. Values are available on both the data flow from +1.KOUT and on the data flow from B.KOUT. Which one should be the output of A? There is nothing in the join free plan which says that +1.KOUT is not the correct value. The join free abstraction process throws away some of the information about how the split PRIME controls the places in which the output of +1 is used. Joins preserve this information and make it easy to determine which value is appropriate. Even in situations where a join free plan would not throw away any information, joins are very useful because they localize in one place all of the information about which value is appropriate when. This advantage is particularly clear in a plan which has many splits and joins.

III.2.3 A Surface Plan

The parts of a plan discussed in this section (III.2) are sufficient to build what is called a "surface plan". A surface plan consists solely of segments (which need not have behavioral descriptions), data flow and control flow. A surface plan (such as the one in Figure III-4) is flat, consisting of an outer segment containing a number of terminal subsegments. The only intermediate segments which are used are those which are necessary in order to express recursion and looping. Given the code for a program, a surface plan can be constructed which describes the physical structure of the program exactly. Section VI.1 discusses a system which can automatically construct surface plans.

A surface plan for a program abstracts away from the syntactic details of the code for the program. However, it is still detailed enough so that it can be executed. All that is needed is an executable function corresponding to each terminal segment in the plan. The data flows in the plan specify what the input values to each segment will be. The order of execution of the segments is controlled by the conditions, data flows, control flows, and the subsegment relation.

The effect of the subsegment arcs on execution order can be modeled by replacing each subsegment arc by a set of control flow arcs, one from each case of the supersegment, going to each case of the subsegment. The effect of the data flow arcs on execution order can be modeled by replacing each data flow arc with a control flow which goes from the side of the case of the segment where the port of origin is located, to the side of the case of the segment where the destination port is located. Once this is done, it can be said that a case of a segment is eligible for execution if and only if its conditions are true, and its control flows are satisfied as follows: the control flows terminating on the case are broken up into a set of groups; each group is either a single control flow or a set of control flows which fan-in disjunctively; the control flows as a whole are satisfied if and only if the source of one control flow in each group has been activated.

There is an assumption of eventual fairness which states that every case of a segment which becomes eligible to be executed will eventually get executed. A plan is said to have its control flow completely specified if it is the case that while it is being executed, there can never be more than one segment which is eligible to be executed, and which has not yet been executed. This is the case in the example above. It should be noted that the definitions above which specify when a segment will be executed are only sufficient in a context where there are no cycles in data flow or control flow, and therefore each segment is executed at most once.

III.3 Describing Why a Segment Does What It Does

This section describes a number of facilities which are used in a plan in order to express the logical structure of a program. They do this by summarizing a proof of correctness for the program. They serve as annotation on a plan constructed with the mechanisms discussed in the previous sections. They do not change what the plan does or how it does it; rather, they help explain why the program works. A plan which includes annotation of its logical structure is referred to as an "annotated", or "complete" plan.

A plan for a program has an outermost segment. The behavioral description for this segment specifies what the program is supposed to do. Verifying that this behavioral description is accurate is equivalent to proving the program correct. The following sections show how such a verification is represented in a plan.

One mechanism which is used is intermediate levels of segmentation. Unlike a surface plan, a complete plan for a program is broken up into a hierarchy of segments within segments, so that each segment has only a few subsegments. Adding this intermediate segmentation does not change the plan's specification of what is going on in the program. What it does do, is provide locality in the plan. It breaks up the problem of understanding the program into a number of sub-problems. It should be noted that it is important to choose a good segmentation of the plan. A random segmentation is no help at all. The PBMs discussed in Chapter IV address the issue of automatically selecting a useful segmentation.

III.3.1 Justification Links

The style of program correctness proof described here verifies a program inductively by proving that the behavioral description of each segment follows solely from the behavioral descriptions of its subsegments and the way the subsegments interact. As a basis step, the behavioral descriptions of the terminal segments are shown to be accurate by reference to the known properties of the primitive functions.

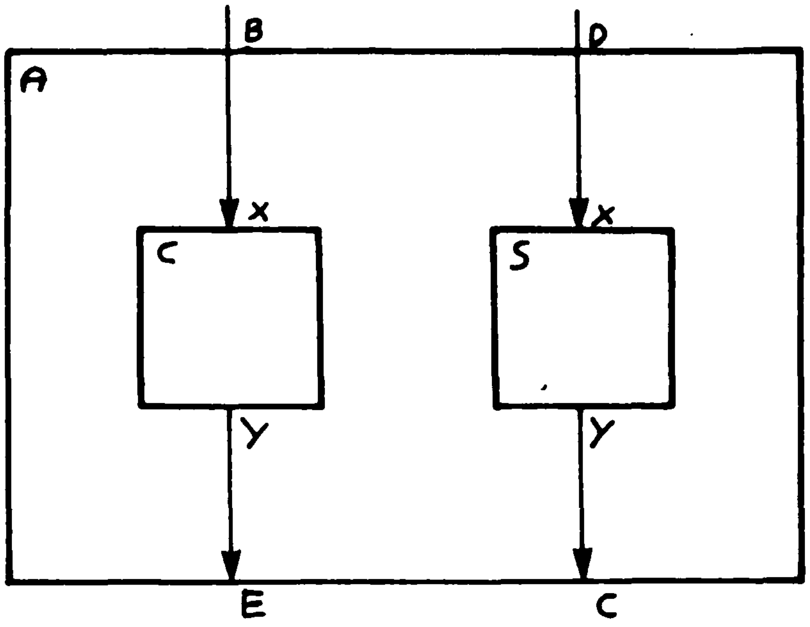
The behavioral description of a non-terminal segment is verified as follows. It is assumed that the behavioral descriptions of its subsegments are correct. It is also assumed that the pre-conditions of the non-terminal segment's behavioral description will be satisfied whenever it is executed. Based on these hypotheses, two things must be proved in order to show that the behavioral description of the non-terminal segment is correct. First, it must be shown that whenever a subsegment is executed, its pre-conditions will be satisfied. Second, it must be shown that the post-conditions in the behavioral description for the non-terminal segment follow from the post-conditions of the subsegments, and the way the subsegments are interconnected by control flow and data flow. The discussion above only addresses the question of partial correctness. In order to gain a complete understanding of a program it is necessary to look at each segment and determine whether or not it will terminate in the situations in which it is executed.

Note that in addition to explicit pre-conditions and post-conditions, each segment has a number of implicit pre-conditions (such as the requirement that there must be a value supplied for each input, and the requirement that the conditions must be mutually exclusive) and implicit post-conditions (such as the claim that each of the outputs actually has a value available when the segment terminates). The justifications for these implicit pre-conditions and post-conditions are usually trivial, and will be omitted from the examples below in the interest of brevity.

The proof method above decomposes the proof of correctness for a program into a set of lemmas to be proved about each segment. There is one lemma associated with each pre-condition of a subsegment, and one lemma associated with each post-condition of a supersegment. The proof of each lemma is summarized in a plan by a "justification link". The justification link for a lemma shows the set of hypotheses on which the conclusion of the lemma depends. These hypotheses include such things as: pre-conditions of the supersegment, post-conditions of subsegments, control flow arcs which specify when a subsegment will be executed, data flow arcs which specify what value a port will receive, and facts in a global knowledge base.

The purpose of the annotation devices being introduced here is to make the logical structure of a program manageable and explicit in its plan. The intermediate segmentation makes the structure manageable by breaking it up into a number of localities which can be dealt with separately. The substitutability of segments guarantees that each individual segment can be understood in isolation from the rest of the plan because no other segment depends on its internal structure, and it does not depend on the internal structure of any other segment. The justification links show how each aspect of the behavioral description of a segment depends on its subsegments, control flow and data flow. Viewed from the other direction, they show the purpose of each feature of a behavioral description of a subsegment, and the purpose of each control flow and data flow. The prime value of the justification links is that they make it easier to reason about the program. For example, suppose that a modification is proposed which removes some features from a plan, and adds some new ones. The justification links immediately tell which things in the plan depended on the features which were removed, and therefore have to be reverified.

The plan in Figure III-5 shows an example of justification links. They are listed on the lines beginning "justifications:". Each justification link is of the form "(a set of supporting facts)→conclusion". Each of the supporting facts and the conclusion consist of a segment name, possibly suffixed by a case identifier, concatenated with a period concatenated with an indicator of the type of the fact followed by a number indicating which specific fact is being referred to. The types can be either: "po" indicating a post-condition, "pr" indicating a pre-condition, "c" indicating a condition, "df" indicating a data flow, "cf" indicating control flow, or "gk" indicating a fact in the global knowledge base. For example, (C.po2, A.d1, A.d2)→A.po2 means that the second post-condition of segment A follows from the second post-condition of segment C, the first data flow of segment A, and the second data flow of segment A.



Flow diagram for Figure III-5

```
                E = COS(B);
                C = SIN(D);

          Segment A (the whole program above)
           inputs: B, D
   pre-conditions: REAL(B), REAL(D)
          outputs: E, C
  post-conditions: REAL(E), E=COS(B), REAL(C), SIN(D)
      subsegments: C, S
        data flow: A.B→C.X, C.Y→A.E, A.D→S.X, S.Y→A.C
   justifications: (C.pol, A.df2)→A.pol, (C.po2, A.dfl, A.df2)→A.po2,
                   (S.pol, A.df4)→A.po3, (S.po2, A.df3, A.df4)→A.po4

          Segment C
           inputs: X
   pre-conditions: REAL(X)
          outputs: Y
  post-conditions: REAL(Y), Y=COS(X)
   justifications: (A.prl, A.dfl)→C.prl

          Segment S
           inputs: X
   pre-conditions: REAL(X)
          outputs: Y
  post-conditions: REAL(Y), Y=SIN(X)
   justifications: (A.pr2, A.df3)→S.prl
```

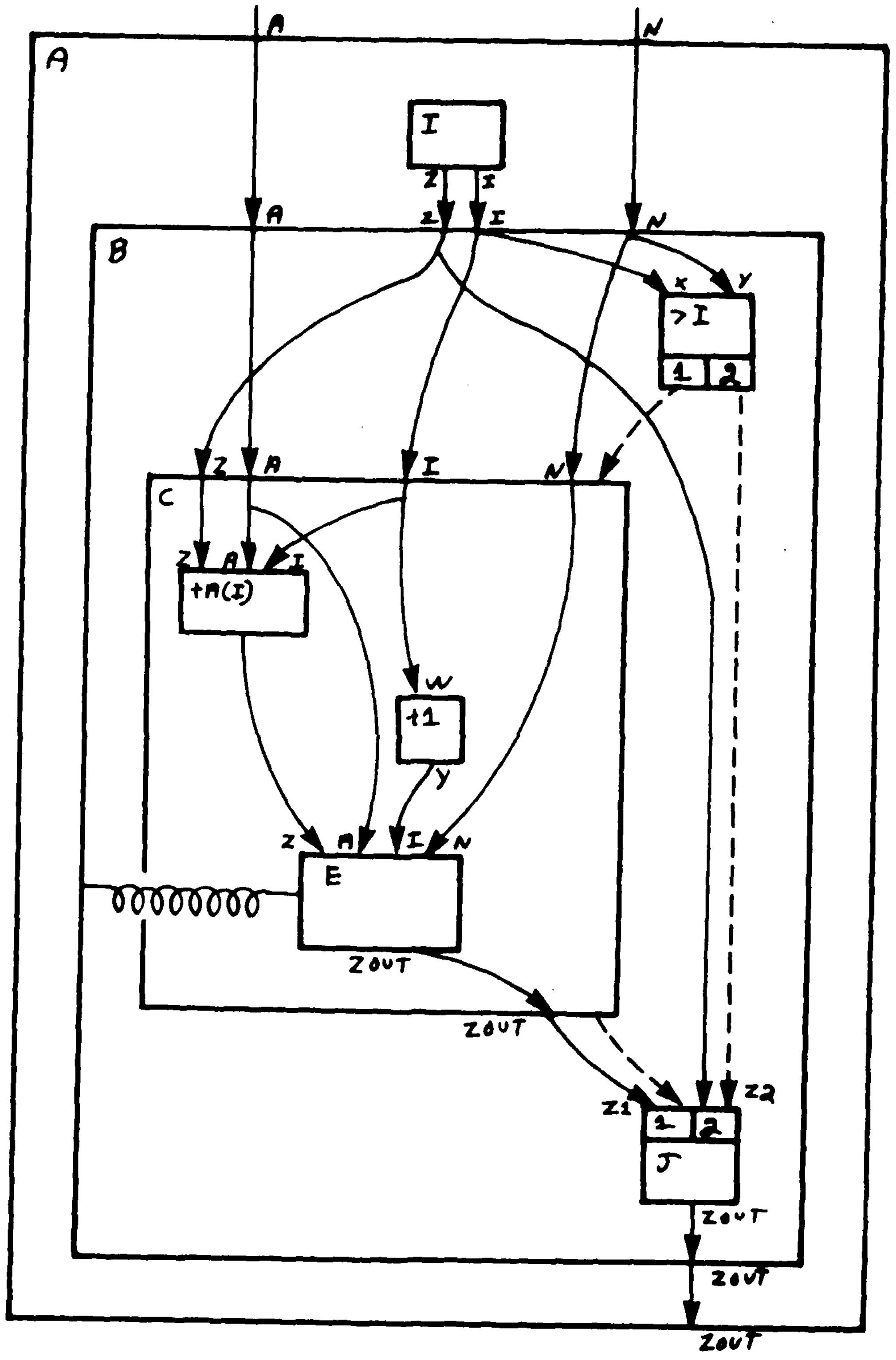
Figure III-5: An example of justification links.

Consider what the justification links signify in the example. The justification link (C.po2, A.df1, A.df2)→A.po2 is a summary of a proof like the following: segment A can assert that E=COS(B) because segment C asserts that Y=COS(X) and data flow in segment A causes X to have the same value as B and E to have the same value as Y. Note that the fact that there are no side-effects in a plan which can contradict the data flow in a plan is an important underpinning of this proof. The justification link (A.pr1, A.df1)→C.pr1 summarizes a proof like the following: the pre-condition REAL(X) of segment C will be satisfied whenever segment C is executed because data flow in segment A causes X to have the same value as input B of segment A and segment A expects REAL(B). Each of the other justification links in the plan is similar to one of the two above. The references to data flows and control flows in justification links will be omitted in the examples below in the interest of simplicity. However, it should be noted that they are an important part of a justification link because they show how the lemma depends on the way the subsegments are interconnected.

Figure III-6 gives a more extended example of an annotated plan. The figure contains a list of global facts which are used in the justification links. These are shown to give an idea of the kind of things that are in the global data base. There are some even more basic global facts which are used in the proof such as: "X=X+0", and "(X+Y)+Z=X+(Y+Z)". It is assumed that facts like these are built into the reasoning system and are never explicitly referred to. This is similar to the way a basic logical relationship, such as substitution of equals for equals, is handled.

There is a recursive link between segments B and E. It should be noted that the recursive link only specifies that the non-annotated plan for E is the same as the non-annotated plan for B. Segment E has justification links which are different from segment B due to the fact that it is situated in a different environment.

```
                 Z = 0;
                 DO I=1 TO N;
                    Z = Z+A(I);
                 END;

         global knowledge used by the justification links
          I>N → $\sum_{j=I,N} A(j) = 0$
          I≤N → $\sum_{j=I,N} A(j) = A(I)+\sum_{j=I+1,N} A(j)$

            Segment A (the whole program above)
             inputs: A, N
     pre-conditions: REAL(A), VECTOR(A), INTEGER(N), N≤SIZE(A)
            outputs: ZOUT
    post-conditions: REAL(ZOUT), ZOUT=$\sum_{j=1,N} A(j)$
        subsegments: I, B
          data flow: A.A→B.A, A.N→B.N, I.Z→B.Z, I.I→B.I, B.ZOUT→A.ZOUT
     justifications: (B.po1)→A.po1, (B.po2, I.po2, I.po4)→A.po2

            Segment I
            outputs: I, Z
    post-conditions: INTEGER(I), I=1, REAL(Z), Z=0
```



Flow Diagram for Figure III-6

```
          Segment B
           inputs: A, I, N, Z
   pre-conditions: REAL(A), VECTOR(A), INTEGER(I), INTEGER(N),
                   REAL(Z), N≤SIZE(A), 1≤I
          outputs: ZOUT
  post-conditions: REAL(ZOUT), ZOUT=Z+∑j=I,N A(j)
      subsegments: >I, J, C
        data flow: B.A→C.A, B.Z→C.Z, B.Z→J2.Z2, B.I→C.I, B.I→>I.X,
                   B.N→C.N, B.N→>I.Y, C.ZOUT→J1.Z1, J.ZOUT→B.ZOUT
     control flow: >I1→C, >I2→J2, C→J1
   justifications: (A.pr1)→B.pr1, (A.pr2)→B.pr2, (I.po1)→B.pr3, (A.pr3)→B.pr4,
                   (I.po3)→B.pr5, (A.pr4)→B.pr6, (I.po2)→B.pr7,
                   (C.po1, J1.po1, B.pr5, J2.po1)→B.po1,
                   (C.po2, >I1.c1, gk2, J1.po1, >I2.c1, gk1, J2.po1)→B.po2

          Segment >I
           inputs: X, Y
   pre-conditions: INTEGER(X), INTEGER(Y)
   justifications: (B.pr3)→>I.pr1, (B.pr4)→>I.pr2
       CASE1
       conditions: X≤Y
       CASE2
       conditions: X>Y

          Segment J
          outputs: ZOUT
       CASE1
           inputs: Z1
  post-conditions: ZOUT=Z1
       CASE2
           inputs: Z2
  post-conditions: ZOUT=Z2

          Segment C
           inputs: A, I, N, Z
   pre-conditions: REAL(A), VECTOR(A), INTEGER(I), INTEGER(N),
                   REAL(Z), N≤SIZE(A), 1≤I≤N
          outputs: ZOUT
  post-conditions: REAL(ZOUT), ZOUT=Z+A(I)+∑j=I+1,N A(j)
      subsegments: +A(I), +1, E
        data flow: C.A→+A(I).A, C.A→E.A, C.I→+A(I).I, C.I→+1.W, C.Z→+A(I).Z,
                   C.N→E.N, +A(I).ZOUT→E.Z, +1.Y→E.I, E.ZOUT→C.ZOUT
   justifications: (B.pr1)→C.pr1, (B.pr2)→C.pr2, (B.pr3)→C.pr3, (B.pr4)→C.pr4,
                   (B.pr5)→C.pr5, (B.pr6)→C.pr6, (B.pr7, >I1.c1)→C.pr7,
                   (E.po1)→C.po1, (E.po2, +A(I).po2, +1.po2)→C.po2

          Segment +A(I)
           inputs: A, I, Z
   pre-conditions: REAL(A), VECTOR(A), INTEGER(I), 1≤I≤SIZE(A), REAL(Z)
          outputs: ZOUT
  post-conditions: REAL(ZOUT), ZOUT=A(I)+Z
   justifications: (C.pr1)→+A(I).pr1, (C.pr2)→+A(I).pr2 (C.pr3)→+A(I).pr3,
                   (C.pr6, C.pr7)→+A(I).pr4, (C.pr5)→+A(I).pr5

          Segment +1
           inputs: W
   pre-conditions: INTEGER(W)
          outputs: Y
  post-conditions: INTEGER(Y), Y=W+1
   justifications: (C.pr3)→+1.pr1
```

```
          Segment E (which has the same non-annotated structure as segment B)
   recursive link: B
           inputs: A, I, N, Z
   pre-conditions: REAL(A), VECTOR(A), INTEGER(I), INTEGER(N),
                   REAL(Z), N≤SIZE(A), 1≤I
          outputs: ZOUT
  post-conditions: REAL(ZOUT), ZOUT=Z+∑j=I,N A(j)
   justifications: (C.pr1)→E.pr1, (C.pr2)→E.pr2, (+1.po1)→E.pr3, (C.pr4)→E.pr4,
                   (+A(I).po1)→E.pr5, (C.pr6)→E.pr6, (C.pr7, +1.po2)→E.pr7
```

Figure III-6: A complex example of a plan with justification links.

The program in the figure is a loop which computes the sum of the first N elements in the vector A. In the plan, the program is represented as a tail recursion. Segment I initializes Z to 0 and I to 1. Segment B corresponds to the repetitive part of the loop. Its behavioral description corresponds to an invariant assertion for the loop. The split >I controls the termination of the loop. Segment C gathers together the computation which is performed on each cycle of the loop. Segment E corresponds to the tail recursive invocation of segment B.

The behavioral description of each of the non-terminal segments (A, B, and C) is justified on the basis of the behavioral descriptions of its subsegments. Consider a few examples of the justification links. The justification link (B.po2, I.po2, I.po4)→A.po2 for the second post-condition of segment A summarizes the following proof: given the appropriate data flow, the post-condition that B computes $ZOUT=Z+\sum_{j=1,N} A(j)$ implies that A computes $ZOUT=\sum_{j=1,N} A(j)$ given that segment I initializes Z to 0 and I to 1.

The justification link (C.po2, >I1.c1, gk2, J1.po1, >I2.c1, gk1, J2.po1)→B.po2 for the second post-condition of segment B summarizes the following proof: there are two situations; in one, the result comes from segment C, and by means of the second global lemma it can be seen that the fact that C computes $ZOUT=Z+A(I)+\sum_{j=I+1,N} A(j)$ implies that B computes $ZOUT=Z+\sum_{j=I,N} A(j)$; in the other situation, ZOUT=Z and the segment >I has determined that I>N, the first global lemma shows that $ZOUT=Z+\sum_{j=1,N} A(j)$ is trivially true.

As a final example, consider the justification link (B.pr7, >I1.c1)→C.pr7 for the seventh pre-condition of segment C. It summarizes a proof like the following: the pre-condition 1≤I≤N of C is guaranteed to be satisfied because segment B expects that 1≤I and segment >I has determined that I≤N.

III.3.2 Describing the Temporal Properties of a Segment

The mechanisms introduced in Section III.1 for describing what a segment does refer only to how the outputs are related to the inputs. They do not make any claims about how the segment performs its computation. The mechanisms presented in this section make it possible to make statements about the internal operation of a segment in its behavioral description. These statements talk about temporal sequences of values created and used by the segment.

The facilities described here are designed to work with recursive programs such as loops. They talk about values in a sequence of states generated by the repeated execution of a part of a program. A basic notion is that of an "execution environment". This gives a name to a control environment corresponding to a place somewhere inside the plan for a segment. The place is identified by singling out a side of a case of a segment. (It is sometimes necessary to put a dummy segment into a plan in order to be able to specify the exact execution environment desired.)

The notion of a "temporal port" is built on the notion of an execution environment. A "temporal output" describes a temporal sequence of values which is available for use in an execution environment. It is specified by indicating a side of a case of a segment and an ordinary port which is available in that environment. A port is available if it is either on the indicated side of the case of the segment, or if it is possible to put in a data flow from the port to the indicated side of the case of the segment.

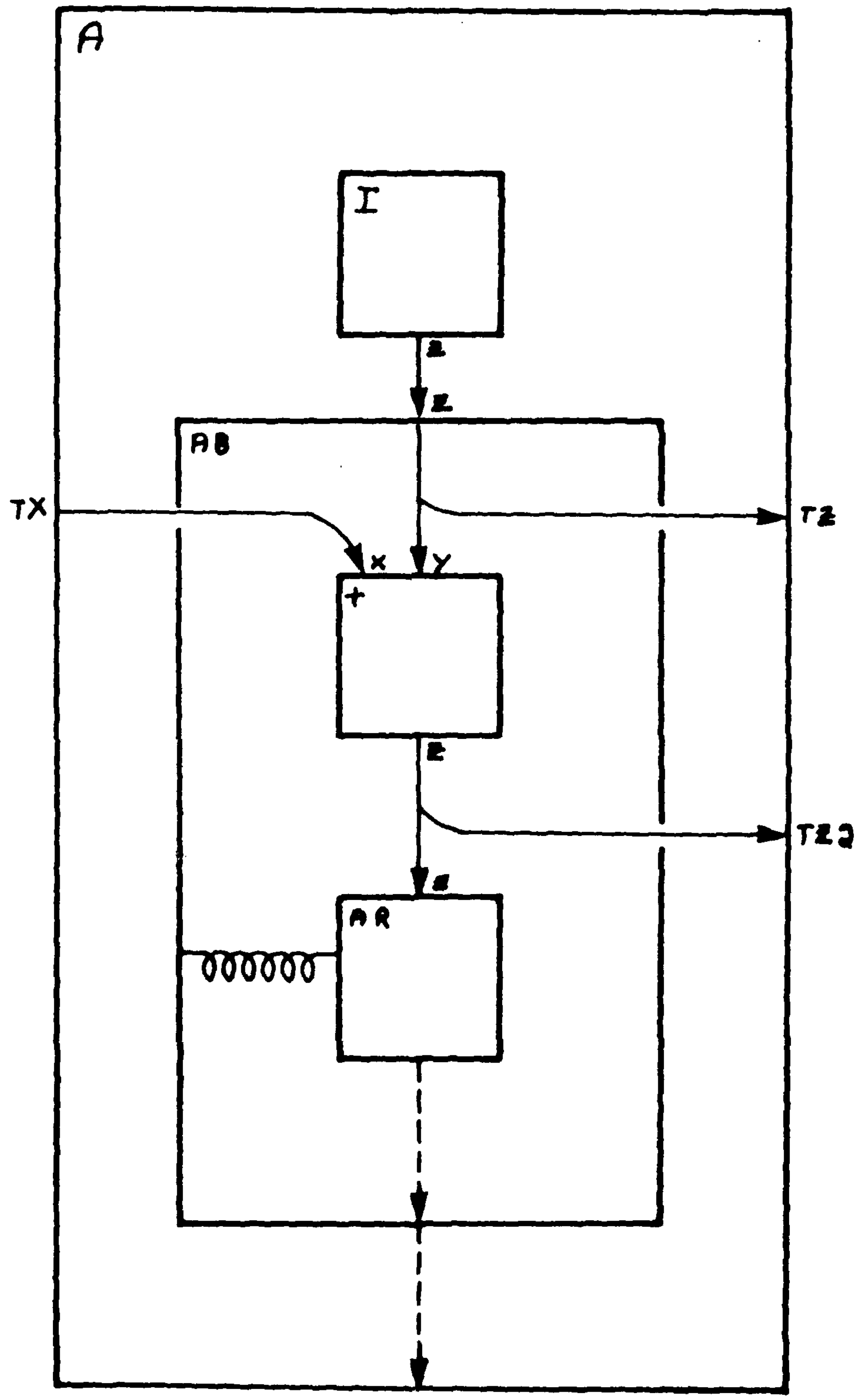
A "temporal input" describes a temporal sequence of values which needs to be supplied in an execution environment. A temporal input is also specified by indicating an ordinary port and an execution environment. With a temporal input the port is required to be on the indicated side of the case of the segment. It must also be the case that there is no data flow going to the port which is associated with the temporal input. The temporal input itself acts like data flow going to the port.

Once a temporal port (say TE) has been given a name, it is possible to make statements about the sequence of states in which the corresponding execution environment is activated, and about the values seen in those states. To facilitate this, the states are mapped onto the integers, with the first state corresponding to 1. A function "last-state" is available for making statements about the number of states in the sequence. "Last-state(TE)=n" means that TE is activated n times. "Last-state(TE)=0" means that TE is never activated.

Figure III-7 shows an example of a segment annotated with temporal information. The line beginning "temporal inputs:" lists the temporal inputs, while the line beginning "temporal outputs:" lists the temporal outputs. Each entry on these lines is of the form "name=(port at segment-case-side)" where "side" is either "i" standing for the input side or "o" standing for the output side. For example, "TX=(+.X at +i)" identifies TX as a name for the sequence of values of the input X of + seen in the sequence of states in which the input side of + is activated.

The flow diagram in the figure represents temporal ports as points on the sides of the segment box. This is in contrast to ordinary ports which appear at the tops and bottoms of the boxes. A temporal port is linked to the ordinary port it is associated with by a data flow arrow. The direction of the arrow indicates whether the port is an input or an output. Note that though the relationship between a temporal port and an ordinary port is graphically represented by a data flow arrow, there is no data flow link between the two in a plan. If the execution environment associated with a temporal port is different from the side of a case of a segment where the ordinary port is located, then the temporal port is linked with its execution environment by a dashed line. This can only happen with a temporal output. There is an example of this in Figure III-8.

```
             Z = 0;
       LOOP: Z = Z+_;
             GOTO LOOP;

            Segment A (the whole program above)
    temporal inputs: TX=(+.X at +i)
     pre-conditions: ∀i∈TX(REAL(TX_i)), last-state(TZ)=last-state(TX)+1
   temporal outputs: TZ=(AB.Z at ABi), TZ2=(+.Z at +o)
    post-conditions: last-state(TZ2)=last-state(TX), ¬terminates(A)
                     ∀i∈TZ(REAL(TZ_i)), ∀i∈TZ2(REAL(TZ2_i)),
                     ∀i∈TZ(TZ_i=∑_{j=1,i-1} TX_j), ∀i∈TZ2(TZ2_i=TZ_{i+1}).
        subsegments: I, AB
          data flow: I.Z→AB.Z
       control flow: AB→A
```



Flow Diagram for Figure III-7

```
          Segment I
          outputs: Z
  post-conditions: REAL(Z), Z=0

          Segment AB
           inputs: Z
   pre-conditions: REAL(Z)
      subsegments: +, AR
        data flow: AB.Z→+.Y, +.Z→AR.Z
     control flow: AR→AB

          Segment +
           inputs: X, Y
   pre-conditions: REAL(X), REAL(Y)
          outputs: Z
  post-conditions: REAL(Z), Z=X+Y

          Segment AR
   recursive link: AB
           inputs: Z
   pre-conditions: REAL(Z)
```

Figure III-7: A segment annotated with temporal information.

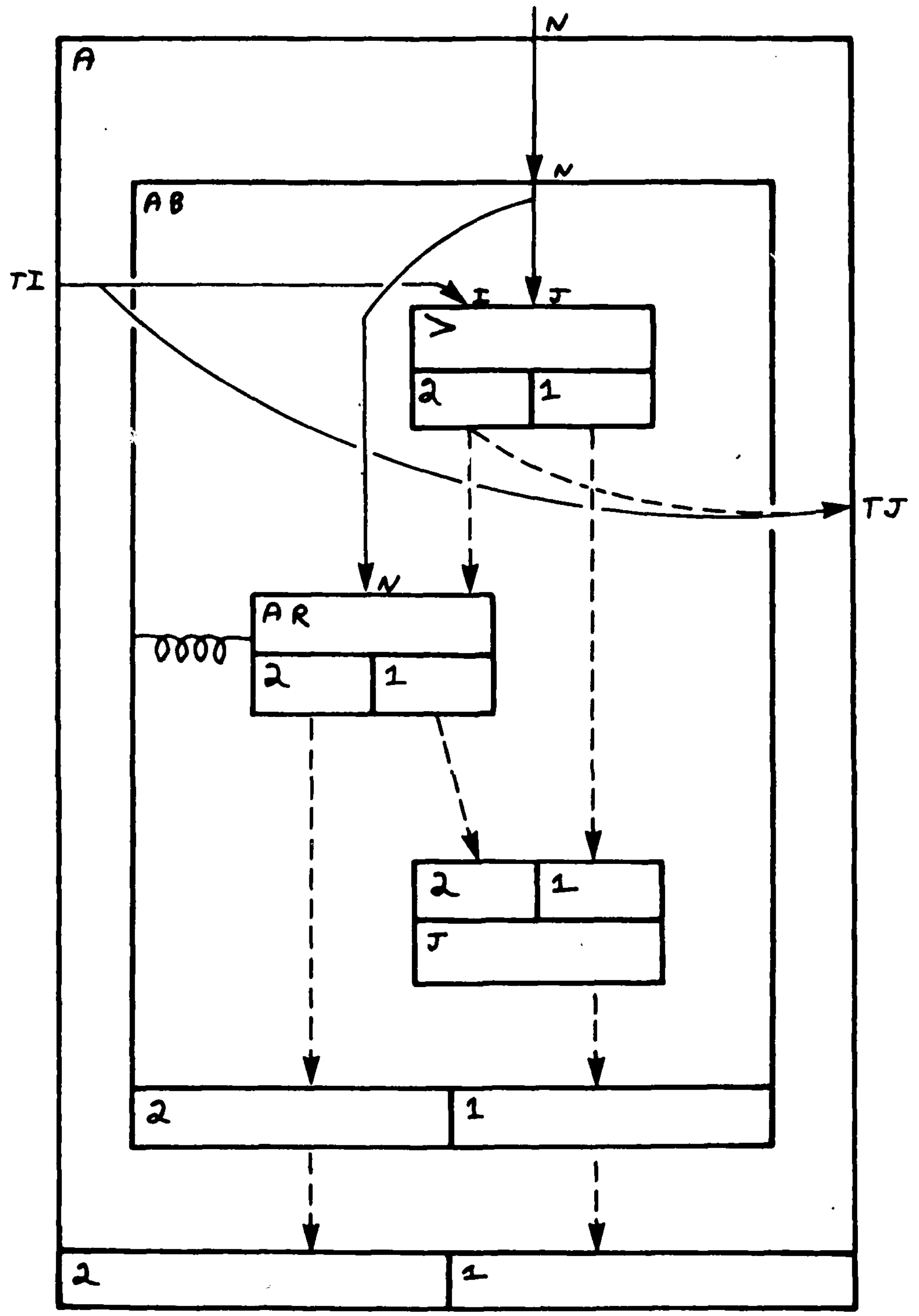
The figure shows the plan for a fragment of a loop. The loop sums up the values supplied to it without ever terminating. The fragmentary nature of the program is indicated by the "_" in the line "LOOP: Z=Z+_". Something has to be put in place of the "_" in order for the program to work. The temporal input TX refers to the situation where the input side of + has just been activated. It represents the temporal sequence of values which must be supplied to + in order for the fragment to operate. (Note: the fact that TX makes a statement about + in the behavioral description for A destroys the substitutability of the segments A and AB. As a result, unlike the other plan notions, the process of temporal description does not treat segments as simply black boxes.) The temporal output TZ refers to the situation where the input side of AB has just been activated. The temporal output TZ2 refers to the situation where + has just completed execution. At that execution environment there could also be a temporal output associated with AB.Z. Both TZ and TZ2 represent temporal sequences of values which are created by the fragment, and which could be used by another fragment.

It is a pre-condition of the plan for segment A that TZ has one more state than TX (note that it is permissible to mention a temporal output in an pre-condition even though it is not permissible to mention an ordinary output). This will be true as long as, if the loop stops, it stops right before an execution of + and not right before an execution of AR, because AB will be activated one time before + is activated the first time, and then it will be activated again (at the input side of AR which is the same as AB) right after each time + is activated. It can be asserted that TZ2 has the same number of states as TX because the execution environments they are associated with are attached to the same case of the same segment. (Note that each segment in a plan is actually only executed once. A temporal port points to a sequence of places, not just one. This has been abbreviated by the recursive link between segments AB and AR. For example, by pointing to +i, TX is actually also pointing to all of the corresponding instances of +i which are abbreviated by the recursive link.)

There is no data flow terminating on the input +.X. As a result, the plan in the figure is actually not well formed and cannot be executed. The temporal input TX corresponds to the sequence of values which needs to be supplied to +.X. Temporal inputs are only associated with ports in a plan which need to have data values supplied to them, but which have no data flow to them in the plan. It is an implicit pre-condition of the temporal input that something must happen so that values are supplied to the internal port. The weakening of constraints on well formed data flow seen in the example is valuable because it allows the specification of plans for fragments which, though not executable, are useful because they can be combined by the PBM temporal composition, and correspond to useful pieces of programming knowledge.

The temporal output TZ corresponds to the sequence of values input to AB.Z. It is clear that the first value of TZ is zero (i.e. $TZ_1=0$). Subscripts are used to identify the values of a temporal port in different states of its temporal environment. It is also clear that "$TZ_i=TZ_{i-1}+TX_{i-1}$" for the value of TZ in all the states of F after the first. This recurrence relation can be solved in order to conclude that "$\forall i \in TZ(TZ_i= \sum_{j=1,i-1} TX_j)$". The expression "$\forall i \in TZ$" refers to all of the states in the execution environment associated with TZ. TZ2 corresponds to the values output by +.Z. These are the same values as TZ with the initial value deleted. The post-condition "¬terminates(A)" states that as it stands, the segment will not terminate. Control can never reach the output side of A. The behavioral description for the loop fragment in the figure is discussed more fully in Section IV.2.1.2.

```
              LOOP: IF _>N THEN GOTO EXIT;
                    GOTO LOOP;
              EXIT: ...

              Segment A (the whole program above)
               inputs: N
     temporal inputs: TI=(>.I at >i)
      pre-conditions: INTEGER(N), ∀i∈TI(INTEGER(TI_i))
    temporal outputs: TJ=(TI_i at >2o)
     post-conditions: ∀i∈TJ(TJ_i=TI_i), ∀i∈TJ(TJ_i≤N)
         subsegments: AB
           data flow: A.N→AB.N
        control flow: AB1→A1, AB2→A2
           CASE1
          conditions: ∃i∈TI(TI_i>N)
     post-conditions: terminates(A), last-state(TI)=MIN({i∈TI| TI_i>N}),
                      last-state(TJ)=last-state(TI)-1
           CASE2
          conditions: ¬∃i∈TI(TI_i>N)
     post-conditions: ¬terminates(A), last-state(TJ)=last-state(TI)

              Segment AB
               inputs: N
      pre-conditions: INTEGER(N)
         subsegments: >, AR, J
           data flow: AB.N→>.J, AB.N→AR.N
        control flow: >2→AR, >1→J1, AR1→J2, J→AB1, AR2→AB2
           CASE1
           CASE2

              Segment >
               inputs: I, J
      pre-conditions: INTEGER(I), INTEGER(J)
           CASE1
          conditions: I>J
           CASE2
          conditions: I≤J
```



Flow Diagram for Figure III-8

```
          Segment AR
recursive link: AB
          inputs: N
pre-conditions: INTEGER(N)
     CASE1
     CASE2

          Segment J
     CASE1
     CASE2
```

Figure III-8: A terminating segment annotated with temporal information.

Figure III-8 shows a loop fragment which is capable of terminating. Here segment A takes a sequence of values via TI and compares them with the ordinary input N. As soon as one of these values is greater than N the loop terminates without looking at any more values of TI_i. The conditions for the two cases of segment A in conjunction with the first post-conditions of these cases state that the loop terminates if and only if there is a value of TI_i which is greater than N.

The temporal output TJ refers to the situation when case 2 of > has just finished execution. TJ passes out the values of TI which are seen in that environment, namely the ones which are less than or equal to N. How many states there are in TJ depends on whether or not the loop terminates. If it does not, then the number of states in TJ is the same as in TI. If it does, then the number of states in TJ is one less than the number of states in TI. The number of states in TI is, in turn, controlled by where in TI the first value greater than N occurs. Note that if TX_1>N then there are no states in TJ. The behavioral description for this fragment is discussed more fully in Section IV.2.1.4.

III.3.3 Representing the Use of PBMs and Temporal Abstraction In a Plan

This section describes how a use of a PBM is represented in a plan. The pieces of information which have to be represented are: what the PBM being used is, and what role is filled by each subsegment. These pieces of information are represented as annotation on the resulting segment. Referring to Figure III-9, the line beginning "pbm:" specifies the PBM (temporal composition in the example) used to build up the segment. The line beginning "roles:" lists the roles of the PBM, and the segments which fill them. Each entry is of the form "role-name role-filler-name". For example, the entry "augmentation1 A1" specifies that the segment A1 fills the role named augmentation1. The PBM and role information is represented in a flow diagram as follows: The name of the corresponding role appears over the upper left corner of each subsegment box. The PBM of the outer box is written above its upper left corner. The purpose of identifying the PBM and roles is to make it easy to apply special purpose methods when working with the segment.

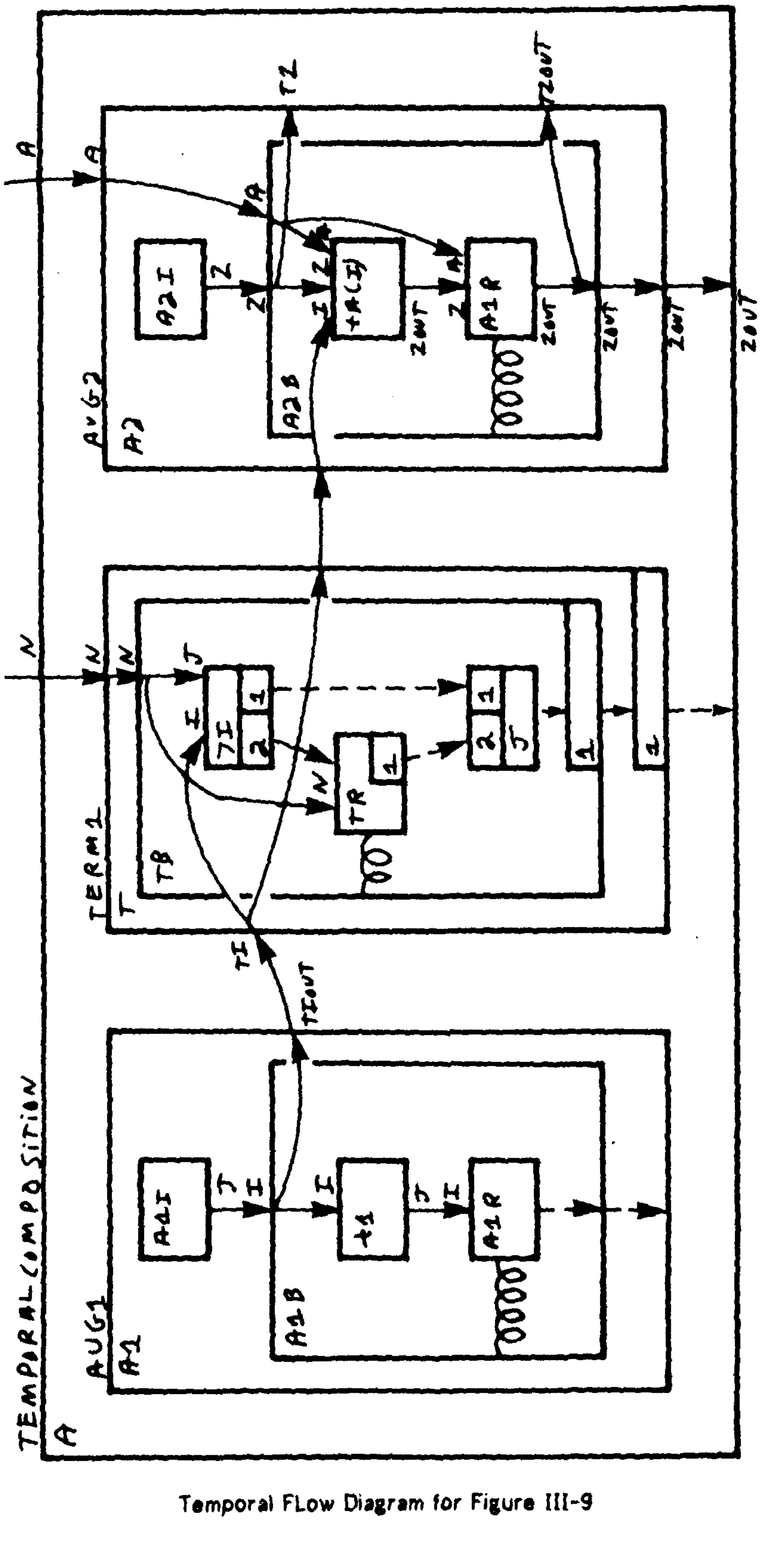
The process of temporal abstraction merely involves taking a segment annotated with temporal information, and treating its temporal inputs and outputs basically just like ordinary inputs and outputs. The figure shows temporal ports connected by data flow which is referred to as "temporal" data flow. The line beginning "temporal dflow:" lists data flow links between temporal ports of the subsegments. (Data flow connecting an ordinary port and a temporal port is not allowed.) Each temporal data flow signifies two things. First it says that the values available at the internal port corresponding to the temporal output will be transmitted to the internal port corresponding to the temporal input. Second, it specifies that the execution environments corresponding to the two temporal ports will be executed in synchrony, the Ith state of one occurring at the same time as the Ith state of the other. Among other things, this implies that the two temporal ports will have the same number of states. For example, the temporal data flow T.TJ→A2.TI indicates that the segment +A(I) will be executed in the same situations that >I2 is executed in, and that the values received by +A(I).I are the same as those received by >I.I.

The figure also shows a temporal flow diagram. A temporal flow diagram is exactly the same as a flow diagram except that the subsegments are temporally abstracted fragments. The temporal data flow is represented by ordinary data flow flow arrows connecting the temporal ports. The inside of each subsegment box is a flow diagram for the subsegment.

```
Z = 0;
DO I=1 TO N;
   Z = Z+A(I);
END;
```

The flow diagram for this program is in Figure III-6.

```
           Segment A (the whole program above)
            inputs: A, N
    pre-conditions: REAL(A), VECTOR(A), INTEGER(N), 0≤N≤SIZE(A)
           outputs: ZOUT
   post-conditions: REAL(ZOUT), ZOUT=∑_{j=1,N} A(j), terminates(A)
         data flow: A.N→T.N, A.A→A2.A, A2.ZOUT→A.ZOUT
      control flow: T1→A
    temporal dflow: A1.TIOUT→T.TI, T.TJ→A2.TI
    justifications: (A2.po2)→A.po1, (A1.po2, A1.po3, T1.c1, T1.po1)→A.po3,
                    (A2.po3, A1.po2, A1.po3, T.po1, T1.c1, T1.po3)→A.po2
               pbm: temporal composition
             roles: augmentation1 A1, termination1 T, augmentation2 A2

           Segment A1
  temporal outputs: TIOUT=(A1B.I at A1Bi)
   post-conditions: ∀i∈TIOUT(INTEGER(TIOUT_i)), ∀i∈TIOUT(TIOUT_i=i), ¬terminates(A1)

           Segment T
            inputs: N
   temporal inputs: TI=(>I.I at >Ii)
    pre-conditions: INTEGER(N), ∀i∈TI(INTEGER(TI_i))
  temporal outputs: TJ=(TI_i at >I2o)
   post-conditions: ∀i∈TJ(TJ_i=TI_i), ∀i∈TJ(TJ_i≤N),
    justifications: (A.pr3)→T.pr1, (A1.po1)→T.pr2
        CASE1
        conditions: ∃i∈TI(TI_i>N)
   post-conditions: terminates(T), last-state(TI)=MIN({i∈TI| TI_i>N}),
                    last-state(TJ)=last-state(TI)-1
        CASE2
        conditions: ¬∃i∈TI(TI_i>N)
   post-conditions: ¬terminates(T), last-state(TJ)=last-state(TI)
```



Temporal FLow Diagram for Figure III-9

```
            Segment A2
             inputs: A
    temporal inputs: TI=(+A(I).I at +A(I)i)
     pre-conditions: REAL(A), VECTOR(A), ∀i∈TI(INTEGER(TI_i)), ∀i∈TI(TI_i≤SIZE(A)),
                     last-state(TZ)=last-state(TI)+1,
                     TZOUT_last-state(TZOUT)=TZ_last-state(TZ)
            outputs: ZOUT
   temporal outputs: TZ=(A2B.Z at A2Bi), TZOUT=(ZOUT at A2Bo)
    post-conditions: ¬terminates(A2), REAL(ZOUT),
                     ZOUT=∑_j=1,last-state(TZ)-1 A(TI_j)
     justifications: (A.pr1)→A2.pr1, (A.pr2)→A2.pr2, (A1.po1, T.po1)→A2.pr3,
                     (A.pr4, T.po2)→A2.pr4, (T)→A2.pr5, (T)→A2.pr6
```

Figure III-9: An example of a plan using the PBM temporal composition.

The program in the figure is the same as the one in Figure III-6. The earlier figure describes the result which is produced when the subsegments of the temporal composition in this figure are combined together eliminating the temporal abstraction (note that the earlier figure is not an instance of the PBM of temporal composition, but of the PBM single self recursion). Comparing the flow diagram in the earlier figure with the temporal flow diagram above shows that they are very different in form. The plan in the figure above represents the program as a composition of segments connected by temporal sequences of values. The plans for the subsegments have been omitted for brevity. They are similar to the examples in the last section. Section IV.2.1.5.2 discusses the plan in the figure in detail. Segment A1 generates the sequence of integers {1,2, ...}. The segment T terminates the loop truncating the sequence of integers at N. The segment A2 takes in this sequence of integers and adds up the corresponding elements of the vector A. The data flow A2.ZOUT→A.ZOUT signifies that the final value of the sum becomes an output of the outer segment.

III.3.4 Verification Based on Temporal Abstraction

Verification of a segment whose plan involves temporal abstraction is performed in the same way, and represented in the same way, as the verification of a segment whose plan does not involve temporal abstraction. The PBMs which deal with temporally abstracted segments are designed so that the proofs which result are correct. It is a basic assumption of a proof based on temporal abstraction that the segments involved are being used in ways which are consistent with the relevant PBMs. As a result, it is not possible to use a proof based on temporal abstraction in a plan which is not annotated with PBM information.

In Figure III-9, the data flow and control flow has the same basic form as a simple composition of segments. As a result, the justification links in the figure follow the basic pattern to be expected of a composition. They are simpler than the justification links in Figure III-6. For example, most of the proof is concentrated in the justification link for the second post-condition of A. This justification link summarizes a proof like the following: A1 enumerates the integers {1,2, ...}, T truncates this sequence at N yielding {1,2, ... N}, A2 sums up the corresponding elements of A yielding "$\mathrm{ZOUT}=\sum_{j=1,N} A(j)$". One of the advantages of this proof is that it does not need to have behavioral descriptions for intermediate segments such as B and C in Figure III-6. Section IV.2 discusses verification involving the PBM temporal composition in greater detail.

III.4 Other Aspects of Plans

There is another dimension to a plan which will not be discussed in detail in this report. This is the dimension of implementation. In the PA, it is envisioned that there will be a hierarchy of plans associated with a given program. At the top will be a very abstract plan for the program. At the bottom is a very concrete plan for the program. The plans in the examples in this report correspond to these most concrete plans.

The process which relates different plans in the hierarchy is implementation. Given an abstract plan, and a more concrete plan for the same program, there are links between them which show what parts of the more concrete plan implement the parts of the more abstract plan. A major purpose of the hierarchy of plans is to encode information about design decisions. For example, the abstract plan might call for the use of a stack, and might have a data flow carrying it. A more concrete plan could embody the design decision that the stack should be implemented using a vector and a stack pointer. In this case, the data flow would have to be implemented as two data flows in parallel, one carrying the vector and one carrying the pointer. The links between the two plans would make the relationship between the two data flows, the design decision, and the more abstract data flow clear.

Another example of implementation is related to the program in Figure III-9. This plan can be looked at as an implementation of a more abstract plan where the main segment is actually just a composition of simple segments, and there is no bias between implementing the program as a loop operating on temporal sequences of values, or implementing it as an actual composition of functions which operate on vectors.

At the lowest level, implementation links show what parts of the code for a program implement each part of the concrete plan for the program. These links are needed in order for the PA to be able to refer to the correct parts of the code when talking with a user about a program.

An important aspect of the relationship between a concrete plan and a program is the use of side-effects. Some of the functions which appear in the code for programs side-effect one of their inputs and return it explicitly, or implicitly, as a result. In this situation, the output is the same data object as the input value with its internal structure altered. This is reflected in a plan by an "id link" between the input port and the output port. Adding this construct into the plan representation brings about a potential conflict with the idea that data flows should be completely unambiguous. To prevent this, it is required that an id link can only be put between an input and an output of a segment, if the data flow and control flow in the plan is such that the data object passed to the input of the segment is never used by any other segment which can be executed later than the segment in question. With this restriction, id links cannot affect data flow in any way, and serve merely as annotation on the plan.

Another important aspect of the implementation hierarchy is that it introduces a second style of verification. A theorem can be proved about a concrete plan by proving it with reference to an abstract plan, and then showing that the concrete plan is a faithful implementation of the abstract plan. This appears as a foundation of the style of proof described above as well. This is true because it calls for proving things about the code for a program with reference to the concrete plan, rather than with reference directly to the code for the program. The proof is only valid because the concrete plan constructed is constrained so that the code is a faithful implementation of it. A greater commonality with an implementation style proof can be seen in the proof based on temporal abstraction discussed in Section III.3.2.3. The proof can be view as an implementation style proof based on an abstract plan where the loop is simply a composition. The PBM has associated with it a lemma proving that the implementation is faithful.

There are additional possible dimensions of plans which have not been investigated yet. An example of this is annotation of efficiency issues. The annotation methods described above are all directed toward explaining why a program works constructed as it is. This annotation does not explain why the program was implemented the way it was instead of some other way. It is clear that understanding efficiency issues is part of completely understanding a program.

It should be noted that the whole area of understanding data structures is intentionally left out of plans. In the PA, data structures are handled by methods similar to data abstractions. Certain aspects of data structures appear in the implementation hierarchy, and it is possible to make arbitrary statements about data structures in pre-conditions and post-conditions. However, plans basically look at all data items as atomic objects without internal structure, and have no features which are explicitly oriented toward dealing with complex data objects. In as much as this is the case, a plan by itself could never embody a complete understanding of a program.

IV. A Catalog of Plan Building Methods

Figure IV-1 indicates the PBMs which have been developed, and how they are interrelated. This chapter discusses each of the PBMs in detail. The structure of the chapter follows the structure of the figure. Section IV.1 describes five PBMs which can be used to analyze non-looping, non-recursive, programs. These PBMs fall into three categories. The PBMs "composition", "conjunction", and "expression" build up arbitrary acyclic combinations of straight-line segments. The PBM "predicate" combines together split segments in order to produce more complex split segments. The PBM "conditional" is closely analogous to the if-then-else construct.

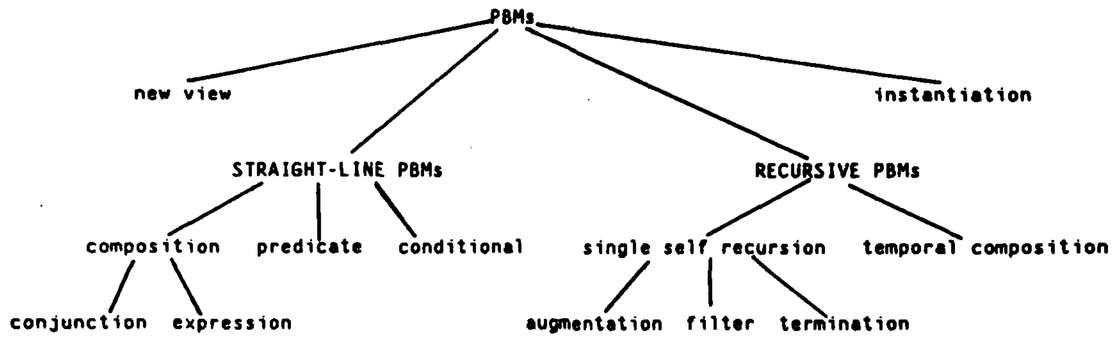


Figure IV-1: A taxonomy of the PBMs.

Section IV.2 describes the PBMs which are used to analyze loops and recursive programs. The PBMs augmentation, filter, and termination are used to construct temporally abstracted fragments of recursive behavior. The PBM temporal composition is used to combine these fragments together.

Section IV.3 discusses the PBM new view which is used to switch to a new point of view when analyzing a program. Section IV.4 describes the meta-PBM instantiation which is used to create a template-like PBM based on some specific plan. Instantiation can be used to analyze a portion of a program as an instance of a known algorithm.

The process of developing PBMs is still continuing. There are interesting classes of programs which are not covered by the PBMs described here. This is discussed more fully in Section IV.5. Chapter V presents an experiment which shows that the PBMs presented here do cover a large class of programs.

Each subsection below follows a similar pattern discussing three major topics. The first topic is how the PBM combines its subsegments in order to produce the resulting segment. As part of this, the subsection describes what roles there are for the PBM, what restrictions there are on the segments which can fill these roles, and what restrictions there are on the control flow and data flow between the subsegments which fill the roles.

The second topic is how a program can be analyzed in terms of the PBM under discussion. It is assumed that the program has been translated into a surface plan. Chapter VI describes a system which does this translation, and performs PBM analysis automatically.

The third topic is how a behavioral description of the resulting segment can be derived and verified based on the behavioral descriptions of its subsegments, and the way the subsegments are interconnected. As part of this, each subsection describes how the subsegments can be looked at in isolation in order to determine a behavioral descriptions for them.

IV.1 Straight-Line PBMs

This section describes the PBMs which can be used to analyze straight-line programs.

IV.1.1 Compositional PBMs

The three PBMs described in this section have a great deal in common. They all combine two or more straight-line segments in order to produce a larger straight-line segment. The roles are all the same, and are called "actions". they are given the names action1, action2, etc. The only basic restriction on the segments which fill the roles is that they must be straight-line segments. The subsegments are connected with each other and to the ports of the resulting segment by an arbitrary acyclic graph of data flow arcs. It is, of course, required that these data flows be well formed as described in Section III.2.2. There are no control flow arcs. The only constraints on execution order are imposed by the data flows. The central restriction on the way in which the subsegments are combined is that there cannot be any loops in data flow.

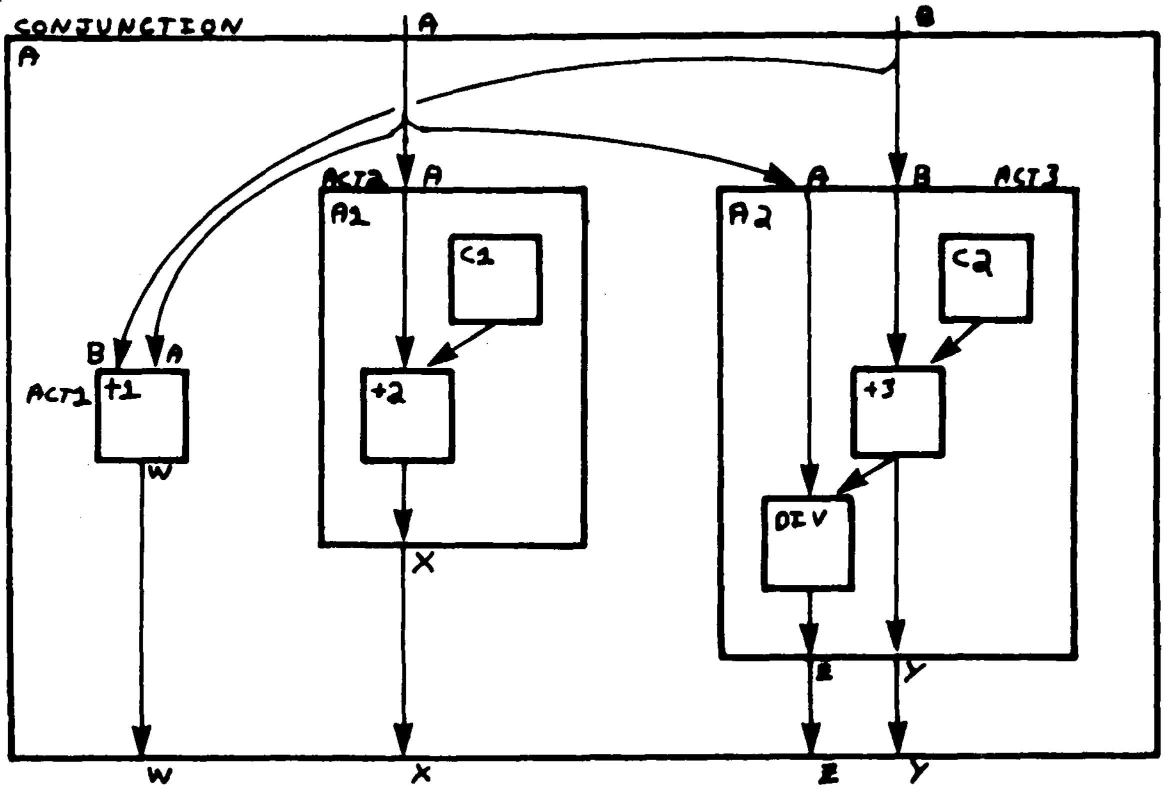
A surface plan for a program can be analyzed in order to find a group of segments which can be understood in terms of one of the compositional PBMs as follows. An appropriate group of segments (which will be referred to as a compositional group of segments) can be found incrementally. Any straight-line segment can be used as a nucleus around which to grow a compositional group. An additional segment can be added to a compositional group whenever either of the following rules apply. If there is a straight-line segment such that it would not contradict anything in the plan if it were executed immediately after the compositional group, then that segment can be added into the group. Similarly if there is a straight-line segment which can be constrained to be executed immediately before the compositional group, then it can be added into the group.

At each stage in the growth of the group, the members of the group are grouped together as the subsegments of an intermediate level straight-line segment. Since the compositional group is itself a straight-line segment the rules above can cause one compositional group to be added into another. If one compositional group contains another, then the segment around the inner one can be eliminated in order to coalesce the two groups. In general, analysis seeks to find maximal compositional groups. Once a compositional group is found, it is further analyzed, as discussed below, in order to see which compositional PBMs apply to it.

IV.1.1.1 The PBM Conjunction

This PBM combines straight-line segments according to the pattern above, with the added restriction that there cannot be any data flow between the subsegments of the result. All of the data flow goes from inputs of the resulting segment to inputs of the subsegments, and from outputs of the subsegments to outputs of the resulting segment. This restriction simplifies the logical structure of the resulting segment. The result simply does the union of all of the things the subsegments do. The PBM conjunction is used by a programmer when he simply wants to do several independent things. Figure III-5 shows a simple example of the PBM conjunction.

When a compositional group of straight-line segments is found, as described above, it is checked in order to see whether it can be analyzed in terms of the PBM conjunction. This is done by partitioning the subsegments of the group into the largest possible number of subsets under the requirement that there must not be any data flow from a subsegment in one subset to a subsegment in a different subset. If this partitioning yields more than one subset, then each subset is grouped into a new intermediate segment and the supersegment is looked at as a conjunction of the new segments. Each of the new segments is a compositional group and is in turn analyzed as described in the next two sections.



Flow Diagram for Figure IV-2

```
                  W = A+B;
                  X = A+1;
                  Y = B+1;
                  Z = A/Y;

          Segment A (the whole program above)
           inputs: A, B
   pre-conditions: REAL(A), REAL(B), B+1≠0
          outputs: W, X, Y, Z
  post-conditions: REAL(W), REAL(X), REAL(Y), REAL(Z),
                   W=A+B, X=A+1, Y=B+1, Z=A/B+1
        data flow: A.A→+1.A, A.A→A1.A, A.A→A2.A, A.B→+1.B, A.B→A2.B,
                   +1.W→A.W, A1.X→A.X, A2.Y→A.Y, A2.Z→A.Z
   justifications: (+1.po1)→A.po1, (A1.po1)→A.po2, (A2.po1)→A.po3,
                   (A2.po2)→A.po4, (+1.po2)→A.po5, (A1.po2)→A.po6,
                   (A2.po3)→A.po7, (A2.po4)→A.po8
              pbm: conjunction
            roles: action1 +1, action2 A1, action3 A2
          Segment +1
           inputs: A, B
   pre-conditions: REAL(A), REAL(B)
          outputs: W
  post-conditions: REAL(W), W=A+B
   justifications: (A.po1)→+1.pr1, (A.po2)→+1.pr2

          Segment A1
           inputs: A
   pre-conditions: REAL(A)
          outputs: X
  post-conditions: REAL(X), X=A+1
   justifications: (A.po1)→A1.pr1

          Segment A2
           inputs: A, B
   pre-conditions: REAL(A), REAL(B), B+1≠0
          outputs: Y, Z
  post-conditions: REAL(Y), REAL(Z), Y=B+1, Z=A/B+1
   justifications: (A.po1)→A2.pr1, (A.po2)→A2.pr2, (A.pr3)→A2.pr3
```

Figure IV-2: An example of the PBM conjunction.

The entire program in Figure IV-2 is a single compositional group. The surface plan for the program is a main segment (A) with six terminal subsegments. These subsegments can be partitioned into three disconnected subsets. These subsets have been grouped together in the figure. One subset has only one element (the segment +1) and therefore does not have to be grouped inside an intermediate segment. After this grouping, segment A is revealed as a conjunction of three subsegments: +1, A1, and A2.

The subsegments are treated like black boxes. They can be completely understood in isolation from each other. When they are combined into the resulting segment, their behavior is not changed in any way. With the conjunction PBM it is particularly easy to develop a behavioral description for the result based on the behavioral descriptions of the subsegments.

Every pre-condition of a subsegment becomes a pre-condition of the result. Similarly all of the post-conditions of the subsegments become post-conditions of the result. The only exception to this is that if some output of a subsegment is not used as an output of the result, then post-conditions referring solely to it need not be brought up to the result. Whenever a logical expression is copied from one segment to another, the inputs and outputs it mentions must be changed to the corresponding inputs and outputs of the new segment. This is due to the fact that a logical expression in the behavioral description of a segment cannot refer to a port of another segment. The correct mapping to use is specified by the data flow between the two segments.

The plan in the figure shows an example of how a behavioral description for the resulting segment can be built up from the behavioral descriptions of its subsegments. The justification links in the plan summarize a verification of the behavioral description for segment A which corresponds to this building up process. It can be seen that the pre-conditions and post-conditions of segment A are just the conjunction of all the pre-conditions and post-conditions of the subsegments.

The only incompatibility problem which can arise when segments are combined according to the conjunction PBM is that the resulting segment could have contradictory pre-conditions. This would happen if two of the subsegments had data flow from the same input of the outer segment, and these two subsegments expected contradictory things about the value. For example, if one assumed it was positive while the other assumed it was negative, or if they expected different types. This would lead to the quite logical result that the resulting segment could never be validly executed since it had unsatisfiable pre-conditions.

It is important to note that there cannot be any conflict between the actions of the subsegments. It is not possible for one subsegment to have a side-effect on some value another subsegment has produced, because if there were such a side-effect, it would be represented in the plan as a data flow between the two subsegments. In that case, the segment could not be analyzed as a conjunction. This is an example of how useful it is that the data flow in a plan is required to accurately reflect the net effect of all side-effects in a program.

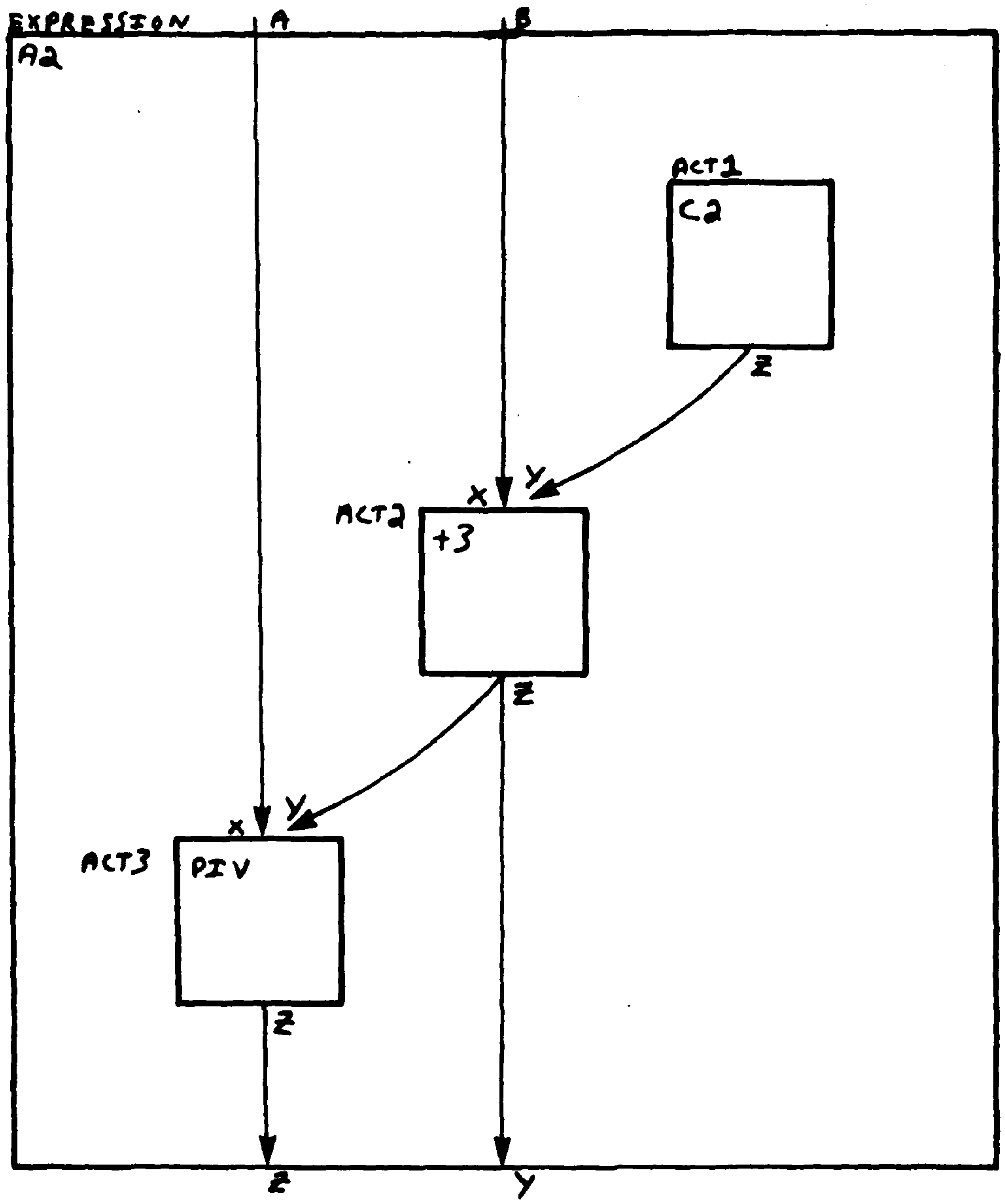
IV.1.1.2 The PBM Expression

This PBM combines straight-line segments according to the pattern described at the beginning of Section IV.1.1 with two added restrictions. First, it must not be possible to analyze the resulting segment as a conjunction. That is to say, it must not be possible to partition the subsegments into disconnected subsets. Second, each subsegment which has data flow going to another subsegment must have a "substitutable" behavioral description.

A behavioral description is called substitutable if each output of the segment is completely described by a substitutable post-condition. A substitutable post-condition is one which is of the form "output=F(inputs)" where "F(inputs)" is an expression which references only inputs of the segment and does not refer to any outputs of the segment. The post-condition "X=(A+B)*C" is a substitutable post-condition (assuming A, B, and C are inputs) while the post-condition "REMAINDER(X,Y)=4" (assuming X and Y are both outputs) is not. The requirement that an output be completely described by such a post-condition means that any other post-condition about the output must have no global importance and can simply be ignored when composing together the behavioral descriptions of the subsegments in order to determine the behavioral description of the result. Type specifications such as "REAL(X)" are examples of post-conditions which are of only local importance. Most of the common functions like +, -, *, etc. have substitutable behavioral descriptions.

When a compositional group of straight-line segments is found, it is first partitioned into disconnected subgroups. The subgroups are then checked to see whether they are expressions. If they are not, then they are treated as compositions (see the next section). The subsegments A1 and A2 in Figure IV-2 are expressions. The plan for A2 is given in Figure IV-3.

```
                  Y = B+1;
                  Z = A/Y;

           Segment A2 (the whole program above)
            inputs: A, B
    pre-conditions: REAL(A), REAL(B), B+1≠0
           outputs: Y, Z
   post-conditions: REAL(Y), REAL(Z), Y=B+1, Z=A/B+1
         data flow: A2.A→DIV.X, A2.B→+3.X, C2.Z→+3.Y, +3.Z→DIV.Y,
                    +3.Z→A2.Y, DIV.Z→A2.Z
    justifications: (+3.po1)→A2.po1, (DIV.po1)→A2.po2 (C2.po2, +3.po2)→A2.po3,
                    (C2.po2, +3.po2, DIV.po2)→A2.po4
          plantype: expression
             roles: action1 C2, action2 +3, action3 DIV
```



Flow Diagram for Figure IV-3

```
         Segment C2
         outputs: Z
post-conditions: REAL(Z), Z=1
```

```
            Segment +3
             inputs: X, Y
     pre-conditions: REAL(X), REAL(Y)
            outputs: Z
    post-conditions: REAL(Z), Z=X+Y
     justifications: (A2.pr2)→+3.pr1, (C2.po1)→+3.pr2

            Segment DIV
             inputs: X, Y
     pre-conditions: REAL(X), REAL(Y), Y≠0
            outputs: Z
    post-conditions: REAL(Z), Z=X/Y
     justifications: (A2.pr1)→DIV.pr1, (+3.po1)→DIV.pr2,
                     (A2.pr3, C2.po2, +3.po2)→DIV.pr3
```

Figure IV-3: An example of the PBM expression.

The program in the figure is a typical expression. All of its subsegments have substitutable behavioral descriptions. Each of the subsegments can be understood in isolation from the rest. A behavioral description for the whole segment is then built up by composing together the pre-conditions and post-conditions of the subsegments. First consider the pre-conditions of the subsegments. Each one is either locally satisfied within the expression or appears as a pre-condition of the outer segment. The pre-conditions of the segment DIV illustrate these two cases. One pre-condition "REAL(Y)" is locally satisfied by a post-condition of the subsegment +3. Another "Y≥0" is propagated up to segment A2 becoming "B+1≥0".

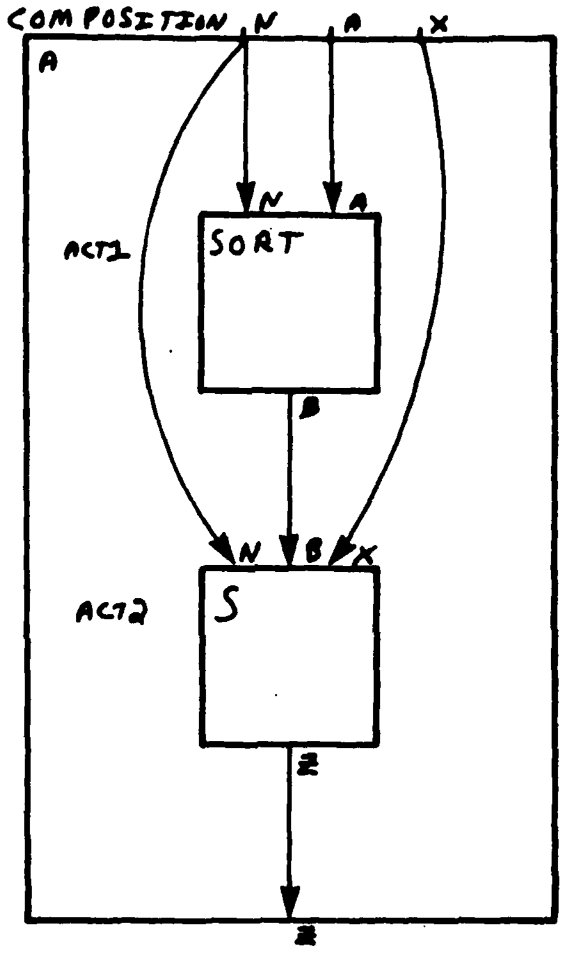
In order to move a pre-condition up to the outer segment, substitutions must be used so that it no longer refers to any internal values. Each reference to an input of a subsegment is eliminated by substituting for it the source of the data flow which terminates on it. Each reference to the output of a subsegment is eliminated by using the substitutable post-condition which describes it. These two operations feed into each other as they proceed. The process is guaranteed to terminate because there are no cycles in the data flow in an expression.

All of the post-conditions about outputs of the subsegments which are carried by data flow to the outer segment are moved up to become post-conditions of the outer segment. The outputs they refer to are replaced by the corresponding outputs of the outer segment. They are then modified by the substitution procedure above so that they do not refer to any internal values. It is interesting to note that if every subsegment of an expression has a substitutable behavioral description, then the behavioral description of the expression will be substitutable.

An expression can have the same incompatibility problems with the pre-conditions of the resulting segment as with a conjunction. In addition it is possible for there to be an internal conflict between the post-conditions of a subsegment which produces a value and the pre-conditions of a subsegment which uses that value. One example of this would be a type conflict.

IV.1.1.3 The PBM Composition

This PBM combines straight-line segments according to the pattern described in the beginning of Section IV.1.1. It is a catchall PBM which is used to analyze any compositional group which cannot be analyzed as a conjunction or expression. A composition is exactly like an expression except that at least one of the subsegments must not have a substitutable behavioral description. This raises a problem when it comes to developing a behavioral description for the outer segment. Quantifiers are used to hypothesize the existence of the internal quantities that the pre-conditions and post-conditions of the outer segment cannot directly refer to.



Flow Diagram for Figure IV-4

```
        CALL SORT(N,A,B);
        DO I=1,N
           IF B(I)>X THEN DO;
              Z=B(I);
              GOTO EXIT;
           END;
        END;
  EXIT: ...
```

```
         Segment A (the whole program above)
          inputs: A, N, X
  pre-conditions: REAL(A), VECTOR(A), INTEGER(N), N=SIZE(A), REAL(X),
                  ∀B((REAL(B) ∧ VECTOR(B) ∧ SIZE(B)=SIZE(A) ∧ PERMUTATION-OF(A,B)
                         ∧ ∀i∀j(1≤i<j≤SIZE(B) → B(i)≤B(j))) →
                     (N=SIZE(B) ∧ ∃i(1≤i≤SIZE(B) ∧ B(i)>X)))
         outputs: Z
 post-conditions: REAL(Z), Z>X,
                  ∃B((REAL(B) ∧ VECTOR(B) ∧ SIZE(B)=SIZE(A) ∧ PERMUTATION-OF(A,B)
                         ∧ ∀i∀j(1≤i<j≤SIZE(B) → B(i)≤B(j))) ∧
                     ∃i(1≤i≤SIZE(B) ∧ B(i)=Z ∧ ∀j(1≤j<i → B(j)≤X)))
       data flow: A.N→SORT.N, A.N→S.N, A.A→SORT.A, SORT.B→S.B,
                  A.X→S.X, S.Z→A.Z
  justifications: (S.po1)→A.po1, (S.po2)→A.po2, (SORT.po1-5, S.po3)→A.po3
             pbm: composition
           roles: action1 SORT, action2 S

         Segment SORT
          inputs: A, N
  pre-conditions: REAL(A), VECTOR(A), INTEGER(N), N=SIZE(A)
         outputs: B
 post-conditions: REAL(B), VECTOR(B), SIZE(B)=SIZE(A), PERMUTATION_OF(A,B),
                  ∀i∀j(1≤i<j≤SIZE(B) → B(i)≤B(j))
  justifications: (A.pr1)→SORT.pr1, (A.pr2)→SORT.pr2,
                  (A.pr3)→SORT.pr3, (A.pr4)→SORT.pr4

         Segment S (the DO loop in the program above)
          inputs: B, N, X
  pre-conditions: REAL(B), VECTOR(B), INTEGER(N), N=SIZE(B),
                  REAL(X), ∃i(1≤i≤SIZE(B) ∧ B(i)>X)
         outputs: Z
 post-conditions: REAL(Z), Z>X, ∃i(1≤i≤SIZE(B) ∧ B(i)=Z ∧ ∀j(1≤j<i → B(j)≤X))
  justifications: (SORT.po1)→S.pr1, (SORT.po2)→S.pr2, (A.pr3)→S.pr3,
                  (A.pr6, SORT.po1-5)→S.pr4, (A.pr5)→S.pr5, (A.pr6, SORT.po1-5)→S.pr6
```

Figure IV-4: An example of the PBM composition.

Consider the composition in Figure IV-4. The two subsegments can be understood in isolation. The process of building up a behavioral description of the outer segment follows the same basic pattern as with an expression. However, a more cumbersome behavioral description results because the output B of SORT is not described by a substitutable post-condition. The pre-conditions of SORT are moved up to segment A. The first and second pre-conditions of segment S are internally satisfied. The third and fifth pre-conditions of S are moved directly up to segment A. The fourth and sixth pre-conditions of S pose a problem. They are not internally satisfied and as a result they must be moved up to segment A. Unfortunately the output B of SORT which they refer to cannot be substituted for. This problem is solved by using a universal quantifier to claim that no matter what the output of SORT is, it must satisfy the post-conditions of SORT and those post-conditions (in conjunction with the pre-conditions of segment A) must ensure that the fourth and sixth pre-conditions of segment S will be satisfied.

A similar problem arises when the post-conditions of segment S are moved up to segment A. The first two can be moved up without any problems. However, the third post-condition refers to the output of SORT. This problem is solved by using an existential quantifier to assert that there must have been some output of SORT, and that this output must satisfy the post-conditions of SORT, and in conjunction with the output of S it must also satisfy the third post-condition of S.

The processes described above can be generalized in order to deal with more complex compositions. The need for these processes is the only difference between a composition and an expression. However, this is a big difference. The problem is that the behavioral description of the result rapidly becomes so cumbersome that it is not good for anything. This problem is aggravated if the result of the composition is then used as a subsegment in another segment.

What is needed with a program like the one in the figure is either a powerful deductive system which can simplify the behavioral description of segment A, or guidance from the programmer as to what the appropriate behavioral description should be. Figure IV-5 shows a more useful behavioral description for segment A.

```
           Segment A
            inputs: A, N, X
    pre-conditions: REAL(A), VECTOR(A), INTEGER(N), N=SIZE(A), REAL(X),
                    ∃i(1≤i≤SIZE(A) ∧ A(i)>X)
           outputs: Z
   post-conditions: REAL(Z), Z=MIN({Y| Y>X ∧ ∃i(1≤i≤SIZE(A) ∧ A(i)=Y)})
```

Figure IV-5: An improved behavioral description for segment A.

This behavioral description is derived by noticing three things. The fourth pre-condition of S "N=SIZE(B)" is actually internally satisfied based on SORT.pr4 ("N=SIZE(A)"), and SORT.po2 ("SIZE(B)=SIZE(A)"). The sixth pre-condition of S "∃i(1≤i≤SIZE(B) ∧ B(i)>X)" will be satisfied by B if and only if there is an element of A greater than X. Finally, the post-conditions of S which claims that Z is greater than X and that Z is the first element of B which is greater than X are the same as saying that Z is the minimum element of A which is greater than X. This is true since B is derived by sorting A in order. It is interesting that this behavioral description for A is actually substitutable.

The deductions above are progressively more difficult. At least the first one is within the reach of any reasonable deductive system. However, this is beyond the trivial deduction system which is assumed to exist in order to do the logical deductions needed by the methods for constructing a behavioral description for a segment. The only thing this minimal deductive system needs is to be able to do the substitution of equals for equals called for by the data flow arcs, and be able to recognize when two logical expressions are identical, that is to say represented by the same sequence of symbols as opposed to equivalent. This is mentioned here in order to show that the methods being described here do not require the presence of any kind of powerful deductive system.

The two special cases of composition, conjunction and expression, are singled out as PBMs in their own right because they occur frequently, and programs analyzed in terms of them can be usefully understood using straight-forward techniques. The division of composition into three PBMs follows the basic idea that the PBMs should be designed so that they separate out the easy parts of a program so they can be attacked with simple methods.

IV.1.2 The PBM Predicate

This PBM combines splits and joins to produce a new segment which is itself a split. The [illegible]pose of the PBM is to build up a complex predicate out of simpler predicates. The PBM has three [illegible] roles. It has one or more splits (named split1, split2, etc.), zero or more terminal joins [illegible] join2, etc.), and an optional initialization (named the initialization). The splits and joins [illegible] to each other and to the cases of the outer segment by an acyclic graph of control flow. The control flow must be well formed and is restricted to be complete as follows. There must be control flows originating on the input side of the outer segment, all of which originate on the NIL case and are therefore the same in every case. There must be control flows terminating on the input side of each split subsegment, all of which terminate on the NIL case. There must be control flows originating on the output side of each join subsegment, all of which originate from the NIL case. Each of the output cases of each of the splits must have a control flow originating from it. Each of the input cases of each of the joins must have a control flow terminating on it. Finally, each of the output cases of the outer segment must have a control flow terminating on it.

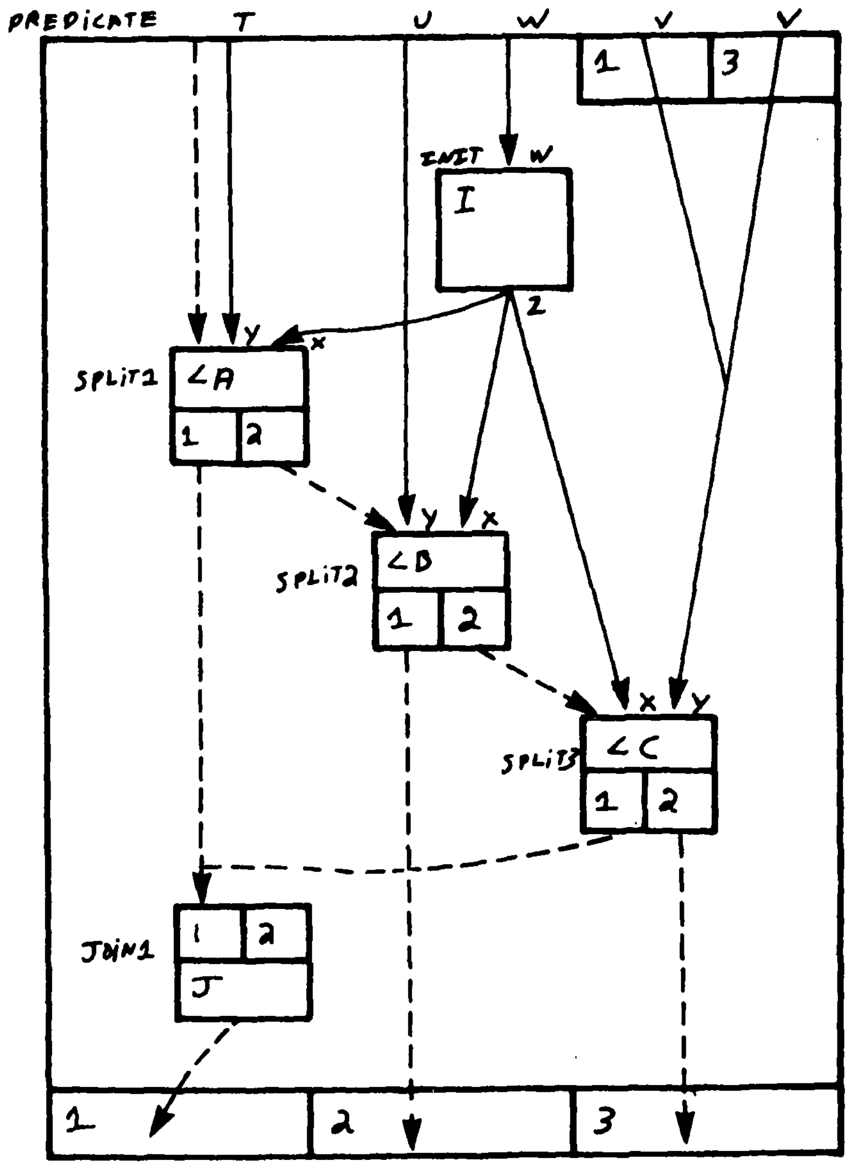
Basically, the splits are connected together like a tree branching out and increasing the number of cases, while the joins act in reverse, collapsing cases together in order to decrease the number of cases. An acyclic graph of data flow connects the inputs of the outer segment with the inputs of the split subsegments. These data flow arcs cannot contradict any of the ordering constraints imposed by the control flow arcs.

The initialization role is a role which is common to most PBMs. It always has the same properties. It must be a straight-line segment with a substitutable behavioral description. There can be data flow from the inputs of the outer segment to inputs of the initialization, but there cannot be any data flow from any other subsegment to the initialization. There is data flow from the initialization to some of the other subsegments, but there cannot be any data flow directly to outputs of the outer segment. Finally, there must not be any control flow to or from the initialization. One of the effects of these restrictions is that the initialization is free to be executed as early as before any other subsegment is executed, or as late as just before the first subsegment which uses one of its outputs.

The basic idea behind an initialization is that it computes some values which are used by the other subsegments and that these values are not used anywhere else but by the other subsegments. Bringing the initialization into the outer segment with the other subsegments usually simplifies the logical structure of the plan for the program containing the segment. In general, if the outputs of the initialization are used by only one subsegment, then the initialization will be moved into that subsegment. The requirement that the behavioral description of the initialization be substitutable and the requirement that the initialization gets all of its inputs directly from inputs of the outer segment means that an initialization will not add any logical complexity to a PBM because any reference to a value produced by the initialization can be eliminated by the substitution of an expression referring only to inputs of the outer segment.

A surface plan can be analyzed in order to detect uses of the PBM predicate as follows. Groups of segments which can be analyzed as predicates are grown by accretion in a way which is similar to the way compositional groups are determined. The process can start with any split segment as a nucleus. From that time on, the group can grow according to the following rules. If there is a split segment whose entering control flow comes directly from the group, then it can be added into the group. If there is a join segment, all of whose entering control flows come directly from the group, then it can be added into the group. If there is a split segment one of whose leaving control flows goes into the group, then it can be added into the group. Note that the group is itself a split segment and therefore it is possible for one group to be added into another. Further if one of the splits which is in a group is itself a predicate, the segment around it can be removed in order to coalesce the two groups. This may or may not be desirable in a given situation.

Once a predicate group has been found, its initialization, if any, can be found by locating straight-line segments which have data flow only directly to the group, which do not have any data flow from the group, and which are described by substitutable behavioral descriptions. Each such segment can be moved into the group's initialization. The resulting initialization will be an expression, or a conjunction of expressions. The procedure outlined above, applies in general to locate the initializations for PBMs which have them.



Flow Diagram for Figure IV-6

```
        X = 2*W;
        IF X<T THEN GOTO L1;
        IF X<U THEN GOTO L2;
        IF X<V THEN GOTO L1;
        ...
    L1: ...
    L2: ...
```

```
         Segment A (the whole program above)
          inputs: T, U, W
  pre-conditions: REAL(T), REAL(U), REAL(W)
       data flow: A.T→<A.Y, A.U→<B.Y, A.W→I.W, I.Z→<A.X, I.Z→<B.X,
                  I.Z→<C.X, A1.V→<C.Y, A3.V→<C.Y
    control flow: A→<A <A1→J1, <A2→<B, <B1→A2, <B2→<C,
                  <C1→J2, <C2→A3, J→A1
  justifications: (<A1.c1, <A2.c1, <B2.c1, <C1.c1)→A1.c1, (<A2.c1, <B1.c1)→A2.c1,
                  (<A2.c1, <B2.c1, <C2.c1)→A3.c1
             pbm: predicate
           roles: split1 <A, split2 <B, split3 <C, join1 J, init I
      CASE1
      conditions: 2*W<T ∨ (2*W≥T ∧ 2*W≥U ∧ 2*W<V)
          inputs: V
  pre-conditions: REAL(V)
      CASE2
      conditions: 2*W≥T ∧ 2*W<U
      CASE3
      conditions: 2*W≥T ∧ 2*W≥U ∧ 2*W≥V
          inputs: V
  pre-conditions: REAL(V)

         Segment I
          inputs: W
  pre-conditions: REAL(W)
         outputs: Z
 post-conditions: REAL(Z), Z=2*W
  justifications: (A.pr1)→I.pr1

         Segment <A
          inputs: X, Y
  pre-conditions: REAL(X), REAL(Y)
  justifications: (I.po1)→<A.pr1, (A.pr1)→<A.pr2
      CASE1
      conditions: X<Y
      CASE2
      conditions: X≥Y

         Segment <B
          inputs: X, Y
  pre-conditions: REAL(X), REAL(Y)
  justifications: (I.po1)→<B.pr1, (A.pr2)→<B.pr2
      CASE1
      conditions: X<Y
      CASE2
      conditions: X≥Y
```

```
             Segment <C
              inputs: X, Y
      pre-conditions: REAL(X), REAL(Y)
      justifications: (I.po1)→<C.pr1, (A1.pr1, A3.pr1)→<C.pr2
           CASE1
          conditions: X<Y
           CASE2
          conditions: X≥Y

             Segment J
           CASE1
           CASE2
```

Figure IV-6: An example of the PBM predicate.

The individual subsegments of a predicate can be understood in isolation. Many splits have the feature that, like the splits in Figure IV-6, they have no data outputs. All of the information which comes out of such a split is encoded in the flow of control. When all of the split subsegments of a predicate have no data outputs, it is particularly easy to build a behavioral description of the outer segment based on behavioral descriptions of the subsegments.

The outer segment has one case for each control flow which goes from a subsegment directly to the output side of the outer segment. The conditions of the cases of the outer segment are boolean combinations of the conditions of the splits. What the conditions for a case are is determined by finding all of the control flow paths through the subsegments which lead to that case. The conditions are the disjunction of the conjunctions of the conditions encountered along each path. The post-conditions of the cases are determined by combining the post-conditions of the splits (if any, there are none in the example) in the same way. Since there are no data outputs of the subsegments the logical expressions in the behavioral descriptions of the splits must all refer only to inputs of the splits all of which come directly from the inputs of the outer segment or from outputs of the initialization. As a result there are no problems with moving these expressions up to the outer segment.

The path analysis reveals which cases each split can be executed in. For example, in the figure the split >3 cannot be executed in the second case. This analysis is used to determine which cases the inputs have to be in. The pre-conditions of each split are moved up to the cases it can be executed in. All of the inputs used by the initialization are put into the NIL case of the outer segment as are the pre-conditions of the initialization. Any references to outputs of the initialization are eliminated by substitution.

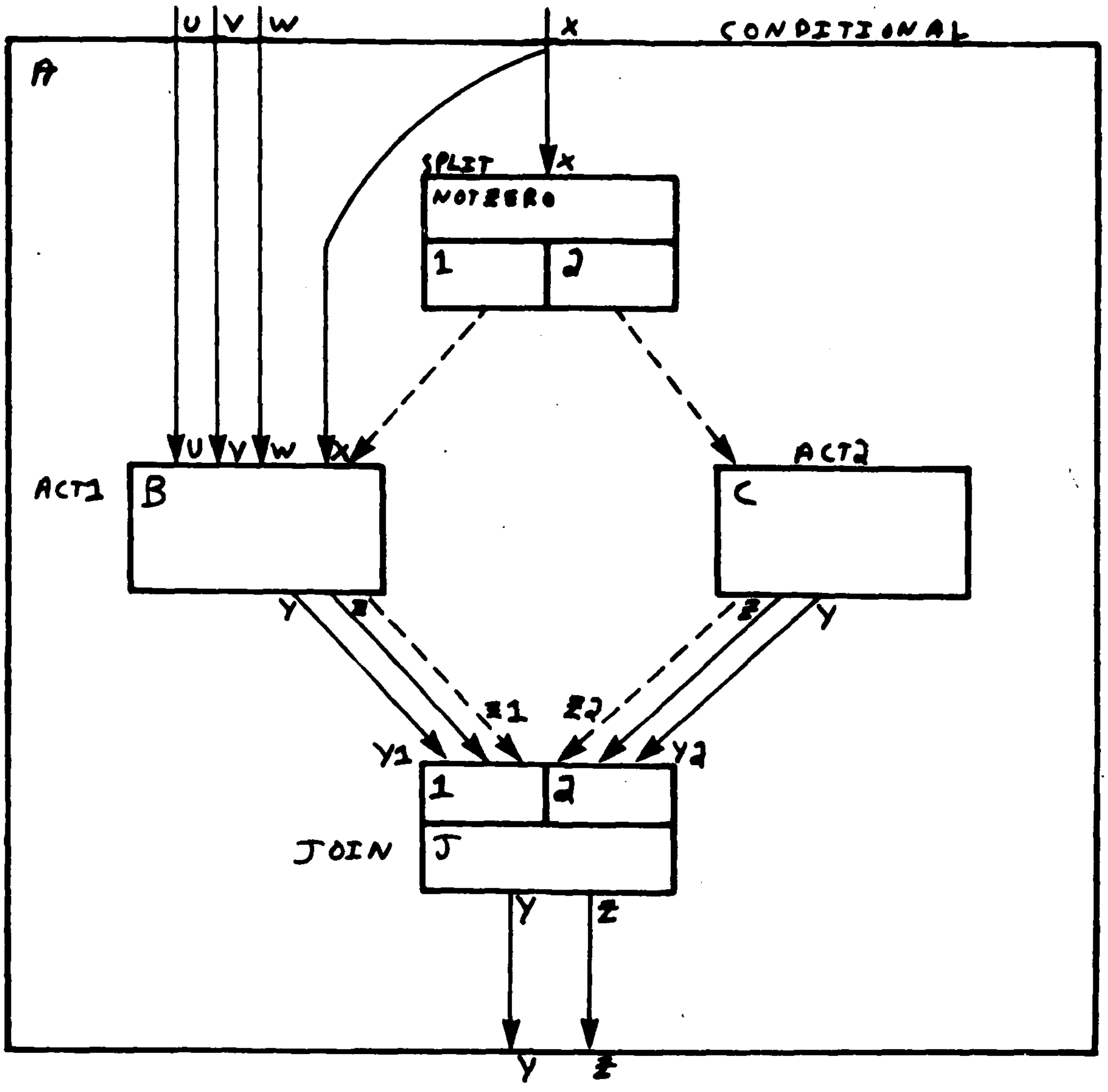
If any of the splits have data outputs, two additional possibilities must be addressed. First, some of these outputs may become outputs of the outer segment. This requires some additional post-conditions to be propagated up to the outer segment. Second, some of the splits may have data flow coming from the outputs of other splits. This is not a problem as long as the sources of these data flows have substitutable behavioral descriptions. If not, the behavioral description of the outer segment may become cumbersome.

IV.1.3 The PBM Conditional

This PBM is analogous to the structured programming construct if-then-else. The PBM has four types of roles. It has one split (named the split). Let M be the number of cases in the split. The PBM has a join (named the join) which must also have M cases. It has zero or more straight-line segments (named action1, action2, etc.) It has an optional initialization (named the initialization) which follows the restrictions for an initialization described in the last section. The resulting segment is a straight-line segment. There must be a control flow path from each case of the split to the corresponding case of the join. There can be zero or one actions on each of these control flow paths.

The control flow in a conditional comes into the split, fans out through the actions, and then fans back in to the join. A surface plan can be analyzed to detect conditionals by looking for this characteristic signature. The initialization can then be found by looking for straight-line segments which have data flow only to the conditional.

```
                  IF X≠0 THEN DO; Z = W/X;   Y = V/U;   END;
                         ELSE DO; Z = 10E70; Y = 10E70; END;

           Segment A (the whole program above)
            inputs: U, V, W, X
    pre-conditions: REAL(X), X≠0→REAL(U), X≠0→REAL(V), X≠0→REAL(W),
                    X≠0→U≠0
           outputs: Y, Z
   post-conditions: REAL(Y), Y=(IF X≠0 THEN V/U  IF X=0 THEN 10E70)
                    REAL(Z), Z=(IF X≠0 THEN W/X  IF X=0 THEN 10E70)
         data flow: A.U→B.U, A.V→B.V, A.W→B.W, A.X→NOTZERO.X, A.X→B.X, B.Y→J1.Y1,
                    B.Z→J1.Z1, C.Y→J2.Y2, C.Z→J2.Z2, J.Y→A.Y, J.Z→A.Z
      control flow: NOTZERO1→B, NOTZERO2→C, B→J1, C→J2
    justifications: (B.po1, J1.po1, C.po1, J2.po1)→A.po1,
                    (NOTZERO1.c1, B.po3, J1.po1, NOTZERO2.c1, C.po3, J2.po1)→A.po2,
                    (B.po2, J1.po2, C.po2, J2.po2)→A.po2,
                    (NOTZERO1.c1, B.po4, J1.po2, NOTZERO2.c1, C.po4, J2.po2)→A.po4
               pbm: conditional
             roles: split NOTZERO, action1 B, action2 C, join J

           Segment NOTZERO
            inputs: X
    pre-conditions: REAL(X)
    justifications: (A.pr1)→NOTZERO.pr1
       CASE1
        conditions: X≠0
       CASE2
        conditions: X=0

           Segment B
            inputs: U, V, W, X
    pre-conditions: REAL(U), REAL(V), REAL(W), REAL(X), U≠0, X≠0
           outputs: Y, Z
   post-conditions: REAL(Y), REAL(Z), Y=V/U, Z=W/X
    justifications: (NOTZERO.c1, A.pr2)→B.pr1, (NOTZERO.c1, A.pr3)→B.pr2,
                    (NOTZERO.c1, A.pr4)→B.pr3, (A.pr1)→B.pr4,
                    (NOTZERO.c1, A.pr5)→B.pr5, (NOTZERO.c1)→B.pr6

           Segment C
           outputs: Y, Z
   post-conditions: REAL(Y), REAL(Z), Y=10E70, Z=10E70
```



Flow Diagram for Figure IV-7

```
           Segment J
           outputs: Y, Z
        CASE1
            inputs: Y1, Z1
   post-conditions: Y=Y1, Z=Z1
        CASE2
            inputs: Y2, Z2
   post-conditions: Y=Y2, Z=Z2
```

Figure IV-7: An example of the PBM conditional.

The conditional in Figure IV-7 has the typical branching out and branching in again structure. The pre-condition "REAL(X)" of the split has become a pre-condition of the outer segment. The pre-conditions of the actions have also been moved up, however, they have been predicated because they only need to be true when their action is performed. Two of these pre-conditions are particularly interesting. The pre-condition "$X \neq 0$" of B is not moved up because it is internally satisfied by the conditions of case 1 of the split. The pre-condition "$U \neq 0$" of B becomes the pre-condition "$X \neq 0 \rightarrow U \neq 0$" of segment A.

There are two fundamentally different ways in which this pre-condition can be satisfied. First, there may be some relationship between X and U so that when X is not zero, U is not zero either. Proving this would show that the pre-condition will be satisfied. Second, U might have nothing to do with X. It might be possible to show that U is not zero whenever segment A is executed. In this case, "$U \neq 0$" would be a more appropriate pre-condition for A. Unfortunately, there is no way to tell which pre-condition is more appropriate when looking at segment A in isolation. As a result, the conservative (i.e. weaker) pre-condition is chosen.

The two ways in which the pre-condition above can be satisfied reflect two different ways in which conditionals are used. Sometimes, the conditional is intimately involved with satisfying the pre-conditions of the actions. This is happening with regard to X in the example. Other times, the conditional simply chooses between two courses of action either of which is applicable. The choice is made on the basis of some higher level goal. An example of this is the conditional "IF MODE=1 THEN M=N*N; ELSE M=N*(N+1)/2;".

Notice that data flow as well as control flow fans in through the join. This reflects the fact that the segment A has only one case. The fan-in represents the fact that the outputs of A come from different actions depending on which one is executed. The post-condition "Z=(IF X≠0 THEN W/X IF X=0 THEN 10E70)" describes the different possible values of Z and which one is produced when. The special expression "(IF X≠0 THEN W/X IF X=0 THEN 10E70)" is used so that segment A will have a substitutable behavioral description. The post-condition is logically equivalent to "$X \neq 0 \rightarrow Z = W/X \wedge X = 0 \rightarrow Z = 10E70$"

In general, when producing the behavioral description for a conditional, the pre-conditions of the split and initialization are simply moved up to the outer segment. The split of a conditional usually does not have any data outputs, and if it does, then they are usually described by substitutable post-conditions. In either of these cases, there is no problem moving the pre-conditions of the actions which are not internally satisfied up to the outer segment. When a pre-condition is moved up, it is predicated with the conditions of the case of the split which has control flow to the action the pre-condition came from. The post-conditions of the actions are moved up to become post-conditions of the outer segment and are similarly predicated. If the actions have substitutable behavioral descriptions, then the logical construct "(IF ... THEN ... . . .)" can be used in order to construct a substitutable behavioral description for the outer segment. If the behavioral description of the split is not substitutable, then things become more complicated in the same way that the PBM composition is more complicated than the PBM expression.

IV.1.4 Analyzing Straight-Line Programs

Taken together, the straight-line PBMs expression, conjunction, composition, predicate, and conditional can be used to build up straight-line programs. Building up a program with these PBMs is equivalent to expressing it in a language where all of the flow of control is expressed by the nesting of expressions, the sequential placement of statements, the calling of subroutines, and the if-then-else construct. Any straight-line program can be expressed in that form. As a result, any straight-line program can be built up using the straight-line PBMs.

Given the code for a straight-line program, it can be analyzed in order to determine how it could have been built up using PBMs. This process is equivalent to translating the code for the program into a language which uses only if-then-else. The difficulty in this is that if the original code for the program was written in a language which has flow of control constructs other than if-then-else (such as pure GOTO) then the code may have to be transformed by duplicating parts of it, or adding additional variables, before it can be translated into a language using only if-then-else. If the code for a program can be expressed using if-then-else without having to be transformed, then it is easy to analyze it in terms of the PBMs whether or not it is actually written using if-then-else. Section VI.2.1 describes a system which performs this analysis automatically. When the code must be transformed, things are more complicated.

The flow diagram in Figure IV-8 shows a prototypical problematical situation, the intersection of two if-then-elses. This can be looked at as either an if-then-else formed of P2, B, C, D, and J2 which is branched into at J1, or as an if-then-else formed of P1, A, B, and J1 which is branched out of at P2. In either case, the diagram cannot be analyzed in terms of the PBM conditional as it stands. There are a variety of ways in which the diagram can be transformed so that it can be analyzed. The segment C can be moved back before the join J1, and duplicated so that it is executed both after A and after B. After this the diagram still cannot be analyzed due to the order of the joins J1 and J2. However, their order can easily be interchanged, so that the diagram can be analyzed as:

```
IF P1 THEN DO; A; C; END;
     ELSE IF P2 THEN DO; B; C; END;
               ELSE D;
```

Alternatively, it may be possible to interchange the order of the tests P1 and P2. This has the advantage that it does not cause any duplication of code. It can be done when case 2 of P2 implies case 2 of P1. This requirement is needed to ensure that the three control flow paths are executed in the correct situations after the transformation. This transformation would cause the diagram to be analyzed as:

```
IF P2 THEN DO; IF P1 THEN A;
                    ELSE B;
               C; END;
     ELSE D;
```

Note that if there had been any computation between P1 and P2 then this transformation could not be applied.

The automatic system described in Chapter VI performs only two kinds of transformations: interchanging of joins, and interchanging of splits. Transformations which duplicate parts of the code or add things to it are not used. One reason for this is that it is very difficult to decide where such transformations should be applied because there are in general a large number of places where they can be applied. The interchanging of joins is handled implicitly by the fact that if two joins are adjacent to each other, they are coalesced together into one big join. The resulting join is not broken apart until a portion of it is analyzed as being part of a predicate or conditional. When two joins are combined, and then separated again, they can be, in effect, interchanged. The interchanging of splits is done explicitly by recognizing situations like the one in the diagram. This is

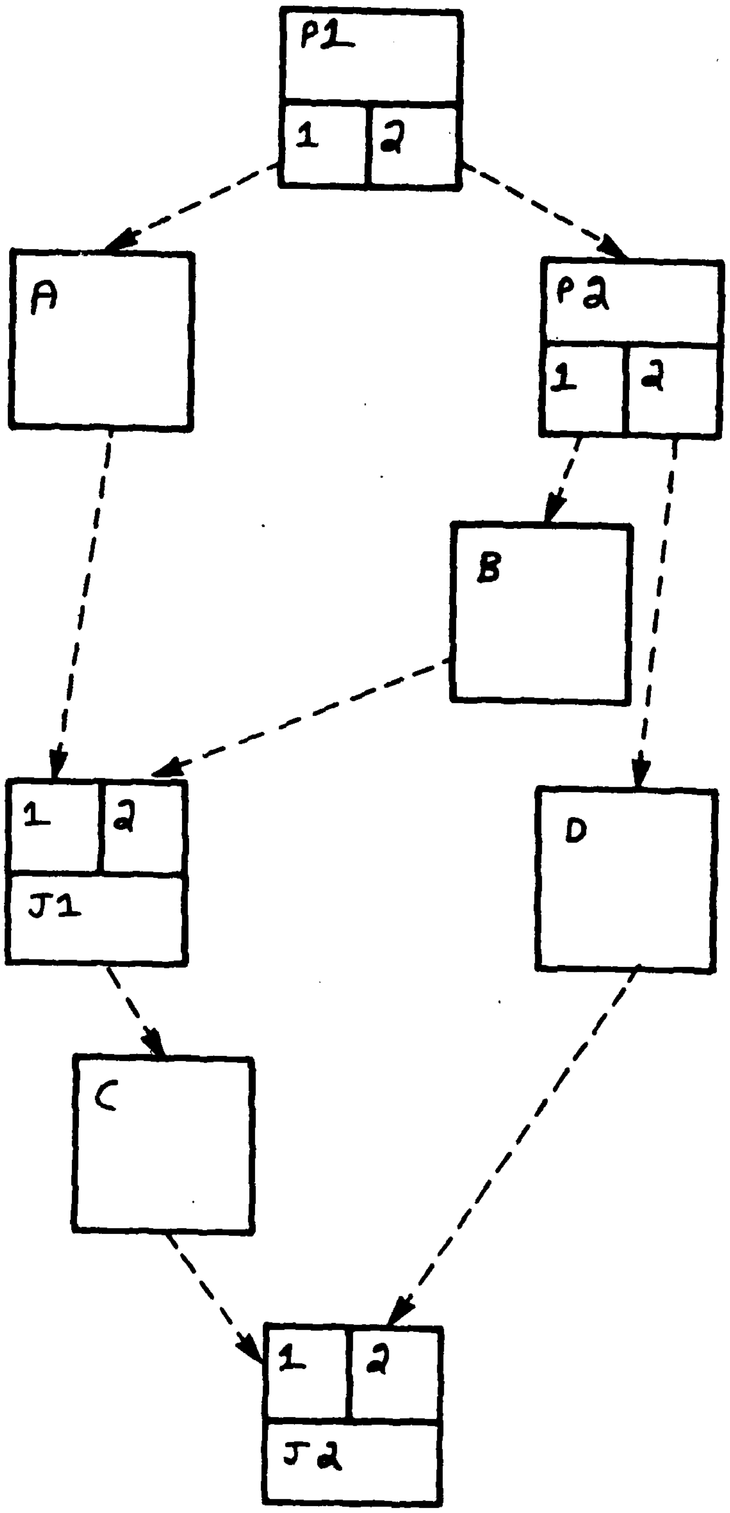


Figure IV-8: An example of intersecting if-then-elses.

discussed further in conjunction with Figure VI-10.

As a result of the limited transformational abilities of the analysis system described in

Chapter VI, it is possible to write the code for a straight-line program in such a way that the system cannot analyze it. For instance, it cannot analyze a program, such as the one below, which has the same basic form as the diagram in the figure and where case 2 of P2 does not imply case 2 of P1.

```
          IF X=0 THEN A;
                 ELSE DO; IF Y=0 THEN DO; D;
                                          GOTO DONE; END;
                          B END;
          C;
    DONE: ...
```

The utility of the analyzer is enhanced by the fact that most programs are relatively straightforward and can be easily analyzed in terms of if-then-elses.

An alternative point of view on the question of transformations comes from the fact that one of the goals of PBM analysis is to uncover the basic logical structure which the programmer had in mind when he wrote the program. From this point of view, a transformation performed by the analyzer is only reasonable if its effect is to undo some transformation which was done by the programmer. For example, it might be the case that in the figure, segment C logically belongs in the two control environments leading to J1, and that the programmer factored it out of those positions, to the position after J1, in order to reduce the size of the code for the program. In that situation, reversing the transformation would be a very reasonable way to analyze the program.

On the other hand, the programmer might have really intended P2 to cause an extraordinary exit from the if-then-else as in the last example program above. In that case, interchanging the two splits might be a reasonable analysis. However, particularly if interchanging the splits is not possible, it might be that no analysis in terms of the PBM conditional captures what the programmer had in mind. In order to really capture the notion of an extraordinary exit, an additional PBM would be required (see Section IV.5).

IV.2 Recursive PBMs

This section describes PBMs which are designed to work with recursive programs. When analyzing a program, iterative looping is translated into tail recursion and analyzed in terms of the recursive PBMs. The recursive PBMs are the most novel and interesting PBMs. They make use of descriptions of the temporal properties of segments (see Section III.3.2).

IV.2.1 Single Self Recursive PBMs

The term "single self recursion" is intended to apply to programs which are recursive in that they call themselves, but which only call themselves from one place inside themselves, and which do not call themselves indirectly through other programs. It should be noted that iterative loops are single self recursive programs and are therefore compatible with the PBMs discussed below. Most of the examples will in fact be loops. Sections IV.2.2 and IV.2.3 discuss how the PBMs could be extended to cover multiple and mutual recursion.

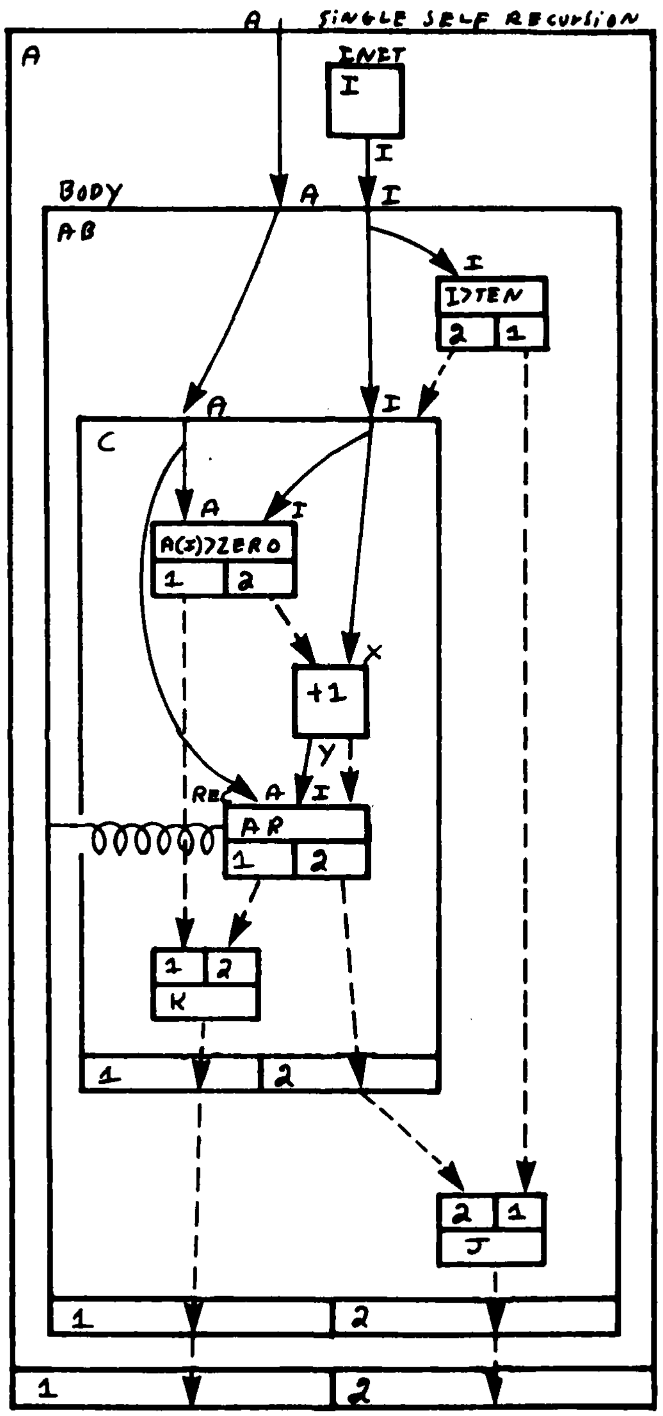
IV.2.1.1 The PBM Single Self Recursion

This PBM is a catch all PBM which constructs arbitrary single self recursive programs. The PBM has three roles: It has an optional initialization (named the initialization) which is a straight-line segment fulfilling the usual requirements on an initialization. It has a body (named the body) which is an arbitrary straight-line segment. It has a recursive instance of the body (named the recurrence) which must be contained in the body, and have the same plan as the body. The only restriction on the structure of the body is that it contain the recurrence.

There can be data flow from the outer segment to the initialization, from the initialization to the body, from the outer segment to the body, and from the body to the outer segment. The body and the outer segment must have the same number of cases. There must be control flow from each case of the body to the corresponding case of the outer segment.

A single self recursion can be annotated with temporal information. However, the PBM is not directly involved with temporal abstraction. In order to produce a non-temporally abstracted surface plan from an instance of single self recursion, the temporal information, if any, is merely deleted.

```
       DO I=1 TO 10;
          IF (A(I)>0) THEN GOTO FOUND;
       END;
       ...
FOUND: ...

          Segment A (the whole program above)
           inputs: A
   pre-conditions: REAL(A), VECTOR(A), 10≤SIZE(A)
        data flow: A.A→AB.A, I.I→AB.I
     control flow: AB1→A1, AB2→A2
              pbm: single self recursion
            roles: initialization I, body AB, recurrence AR
       CASE1
       conditions: ∃i(1≤i≤10 ∧ A(i)>0)
       CASE2
       conditions: ∀i(1≤i≤10 → A(i)≤0)

          Segment I
          outputs: I
  post-conditions: INTEGER(I), I=1

          Segment AB
           inputs: A, I
   pre-conditions: REAL(A), VECTOR(A), 10≤SIZE(A), INTEGER(I)
      subsegments: I>TEN, C, J
        data flow: AB.A→C.A, AB.I→C.I, AB.I→I>TEN.I
     control flow: I>TEN1→J1, I>TEN2→C, C1→AB1, C2→J2, J→AB2
       CASE1
       conditions: ∃i(I≤i≤10 ∧ A(i)>0)
       CASE2
       conditions: ∀i(I≤i≤10 → A(i)≤0)
```



Flow Diagram for Figure IV-9

```
          Segment I>TEN
           inputs: I
   pre-conditions: INTEGER(I)
      CASE1
       conditions: I>10
      CASE2
       conditions: I≤10

          Segment C
           inputs: A, I
   pre-conditions: REAL(A), VECTOR(A), I≤SIZE(A), INTEGER(I)
      subsegments: A(I)>ZERO, +1, AR, K
        data flow: C.A→A(I)>ZERO, C.A→AR.A, C.I→A(I)>ZERO.I,
                   C.I→+1.X, +1.Y→AR.I
     control flow: A(I)>ZERO1→K1, A(I)>ZERO2→+1, +1→AR,
                   AR1→K2, AR2→C2, K→C1
      CASE1
       conditions: A(I)>0 v ∃i(I+1≤i≤10 ∧ A(i)>0)
      CASE2
       conditions: A(I)≤0 ∧ ∀i(I+1≤i≤10 → A(i)≤0)

          Segment A(I)>ZERO
           inputs: A, I
   pre-conditions: REAL(A), VECTOR(A), I≤SIZE(A), INTEGER(I)
      CASE1
       conditions: A(I)>0
      CASE2
       conditions: A(I)≤0

          Segment +1
           inputs: X
   pre-conditions: INTEGER(X)
          outputs: Y
  post-conditions: INTEGER(Y), Y=X+1

          Segment AR
   recursive link: AB
           inputs: A, I
   pre-conditions: REAL(A), VECTOR(A), 10≤SIZE(A), INTEGER(I)
      CASE1
       conditions: ∃i(I≤i≤10 ∧ A(i)>0)
      CASE2
       conditions: ∀i(I≤i≤10 → A(i)≤0)

          Segment J
      CASE1
      CASE2

          Segment K
      CASE1
      CASE2
```

Figure IV-9: An example of the PBM single self recursion.

Figure IV-9 shows an example of a program constructed by the single self recursion PBM. It follows the typical pattern. The outer segment A contains the initialization (I) which is executed once, and the body (AB) which is executed repeatedly. The recurrence (AR) is inside the body. Note, that it is inside segment C which is inside the body AB. There is no limit to how far down inside the body the recurrence can be. In the example, the body and the outer segment are both splits. In case 1, the loop halts because it has found a positive element in A. In case 2, the loop

AD-A084 818 MASSACHUSETTS INST OF TECH CAMBRIDGE ARTIFICIAL INTE--ETC F/G 9/2
AUTOMATIC ANALYSIS OF THE LOGICAL STRUCTURE OF PROGRAMS.(U)
DEC 78 R C WATERS N00014-75-C-0643
UNCLASSIFIED AI-TR-492 NL

2 OF 3
AD
A084818

halts after looking at the first 10 elements of A and discovering that none of them are positive.

Figure III-6 gives another example of a single self recursion. In that example, the initialization is I, the body is B and the recurrence is E. The body (B) computes a sum and is not a split. It should be noted that with the single self recursion PBM, there is no limit on the complexity of the body. Both of the examples discussed above are tail recursive, and can therefore be implemented as loops. This is equivalent to saying that there is no computation which is performed after the recurrence in the body. Note that joins do not perform any computation.

A program can be analyzed in order to find instances of single self recursion by noticing instances of recursion and looping (which is converted to recursion). Once an instance of recursion is detected, the segment which is called recursively is checked to see that it calls itself only directly, and only once. If it does, then it is the body of a single self recursion. The place where it calls itself is the recurrence. The initialization is found like any other initialization by locating segments which have data flow to, and only to, the body. The initialization is not put inside the body, because that would imply that it should be executed more than once.

Except for the fact that the recurrence is inside the body, the PBM single self recursion is just like a straight-line PBM. The only techniques associated with it for determining a behavioral description for the result are based on combining behavioral descriptions for the initialization and body. If there are behavioral descriptions for the body and the initialization, then tnere is no problem. The single self recursion can be treated as either a very simple predicate (as in the example above), or as a very simple composition (as in Figure III-6). If there were justification links in the example above, they would be very similar to those in Figure III-6.

The problem with this is that in practical situations, the body (and hence the recurrence) usually does not have a complete behavioral description. It is easy to determine what its inputs and outputs are, but it is not easy to determine what its pre-conditions, post-conditions, or conditions are. The straight-line PBMs discussed in Section IV.1 are no help. They require that all of their subsegments have complete behavioral descriptions. They cannot be applied in the body of a single self recursion unless the recurrence has a complete behavioral description, which it does not when the body does not. The result of all this is that the single self recursion PBM is not very helpful when it comes to determining a behavioral description for the result.

The three PBMs below (augmentation (Section IV.2.1.2), filter (Section IV.2.1.3), and termination (Section IV.2.1.4)) correspond to restricted types of single self recursive programs. With each of them, the body is simple enough so that general methods can be applied to determine the behavioral description of the result, without having a behavioral description for the body to start with. These methods look inside the body and take special account of the fact that the body contains an instance of itself.

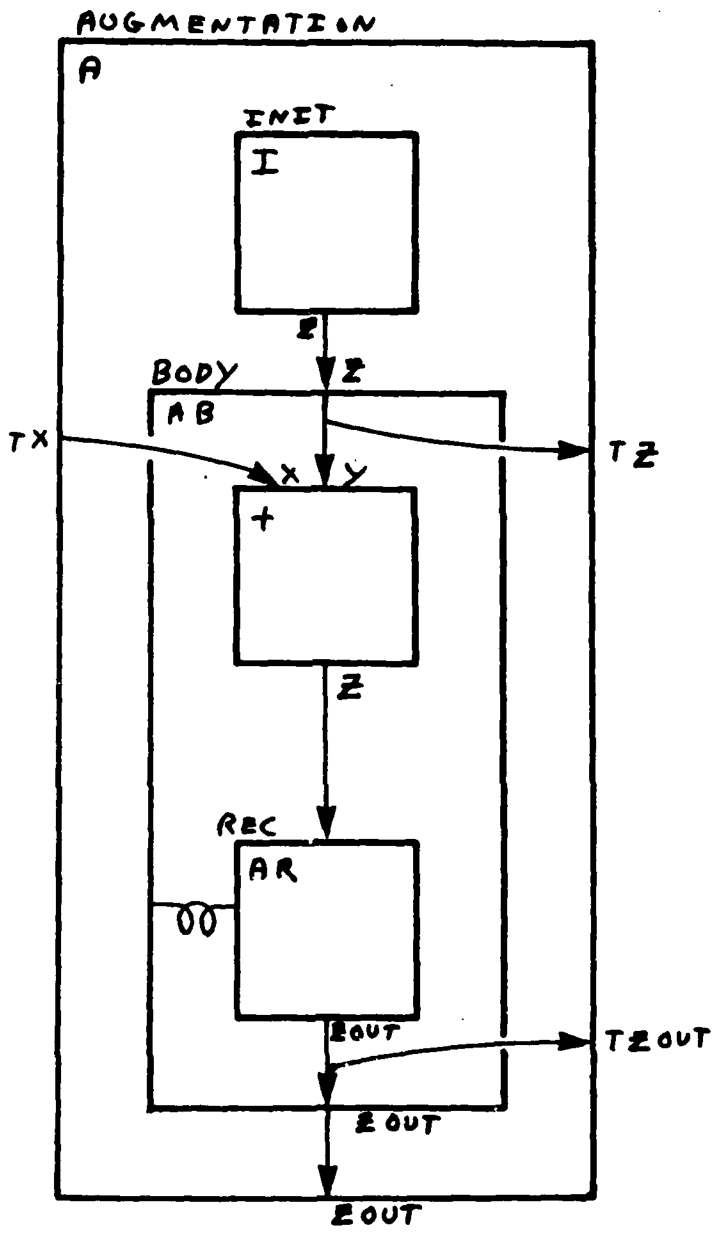
These three PBMs go a long way toward solving the problem of determining a behavioral description for the result in simple cases. However, they cannot be applied directly to realistically complicated programs. The PBM temporal composition (Section IV.2.1.5) can be used to break a complex single self recursive program apart into simpler pieces which can be dealt with by the PBMs augmentation, filter, and termination. The pieces created by the temporal composition PBM are annotated with temporal information and are usually fragmentary in that they have missing data flow inside them. The next three sections discuss how these fragments can be understood in isolation. The section after that talks about how they can be put together.

IV.2.1.2 The PBM Augmentation

The augmentation PBM is the same as the single self recursion PBM, having the same three roles (initialization, body, and recurrence), except for some additional restrictions on the body. The body of an augmentation must have only one case. It must have only two subsegments. One of these must be the recurrence and the other must be some other straight-line segment which will be referred to as the "augmentation function". The augmentation function must be constrained to be executed either always before the recurrence (in which case the augmentation will be called tail recursive) or always after the recurrence (in which case the augmentation will be called head recursive). One consequence of these restrictions on the body is that it can never terminate execution. There can be temporal outputs referring to the inputs and outputs of the body, the recurrence, and the augmentation function. However, there cannot be any referring to the interior of the augmentation function. There can be temporal inputs referring only to the inputs of the augmentation function.

The first thing to notice about augmentations is that due to temporal inputs referring to internal ports which are not satisfied by data flow, they are usually not well formed and therefore cannot actually be executed. Even when they can be executed, they can never terminate. As a result, naked augmentations seldom occur in a program. They are useful because they can be simply understood, and they can be used as building blocks by the PBM temporal composition in order to produce the kinds of loops and recursions found in actual programs. This section discusses augmentations and how they can be understood in isolation. Section IV.2.1.5 discusses how augmentations can be combined together and how the resulting segments can be understood in terms of the augmentations they are built out of. The name augmentation comes from the fact that an augmentation can be added into a recursive program in order to produce a larger (augmented) program which performs some additional computation, without changing anything the original program was doing.

Augmentation is recognized as a special case of single self recursion. Once a program is analyzed as a single self recursion, it can be checked to see whether the additional restrictions on the body are satisfied. Single self recursions can be detected as discussed in the last section. In addition Section IV.2.1.5 discusses how a large single self recursion can be broken up into a combination of simpler single self recursions.



Flow Diagram for Figure IV-18

```
                 Z = 0;
           LOOP: Z = Z+_;
                 GOTO LOOP;

           Segment A (the whole program above)
 temporal inputs: TX=(+.X at +i)
  pre-conditions: ∀i∈TX(real(TX_i)), last-state(TZ)=last-state(TX)+1,
                  TZOUT_last-state(TZOUT)=TZ_last-state(TZ)
temporal outputs: TZ=(AB.Z at ABi), TZOUT=(AB.ZOUT at ABo)
         outputs: ZOUT
 post-conditions: ¬terminates(A), ∀i∈TZ(REAL(TZ_i)),
                  ∀i∈TZ(TZ_i=Σ_j=1,i-1 TX_j),
                  REAL(ZOUT), ZOUT=Σ_j=1,last-state(TZ)-1 TX_j
       data flow: I.Z→AB.Z, AB.ZOUT→A.ZOUT
  justifications: (+.po1, I.po1)→A.po2, (I.po2, +.po2)→A.po3,
                  (A.pr3, A.po2)→A.po4, (A.pr3, A.po3)→A.po5
             pbm: augmentation
           roles: initialization I, body AB, recurrence AR

         Segment I
         outputs: Z
 post-conditions: REAL(Z), Z=0

         Segment AB
          inputs: Z
  pre-conditions: REAL(Z)
         outputs: ZOUT
     subsegments: +, AR
       data flow: AB.Z→+.Y, +.Z→AR.Z, AR.ZOUT→AB.ZOUT
  justifications: (I.po1)→AB.pr1

         Segment +
          inputs: X, Y
  pre-conditions: REAL(X), REAL(Y)
         outputs: Z
 post-conditions: REAL(Z), Z=X+Y
  justifications: (A.pr1)→+.pr1, (AB.pr1)→+.pr1

         Segment AR
  recursive link: AB
          inputs: Z
  pre-conditions: REAL(Z)
         outputs: ZOUT
  justifications: (+.po1)→AR.pr1
```

Figure IV-10: An example of an augmentation.

The plan in Figure IV-10 shows an example of a tail recursive augmentation which sums up a sequence of numbers. This is essentially the same plan as in Figure III-7. A key question which was not addressed in detail in the discussion of the earlier figure is how the behavioral description of the outer segment can be derived. One part of this is how are pre-conditions and post-conditions for ordinary inputs and outputs derived. Another is how are pre-conditions and post-conditions for temporal ports derived. Note that to start with, the only segments which have any pre-conditions or post-conditions are the initialization (I) and the augmentation function (+). The resulting segment (A), the body (AB) and the recurrence (AR) do not.

The pre-condition "REAL(Y)" for the input Y of + is moved up to segment AB. The resulting pre-condition "REAL(Z)" is internally satisfied by a post-condition of segment I. This pre-condition of AB also appears as a pre-condition of AR since AR is recursively linked to AB. At AR, the pre-condition is internally satisfied by a post-condition of + itself. The fact that the output of + becomes an input of + on the next iteration, reflects a common feature of augmentations.

Assertions about temporal ports can be developed as follows: There are only six places in an augmentation which can be identified as execution environments (in the example: TZ at ABi, TX at +i, at +o, at ARi, at ARo, and TZOUT at ABo). Places which refer to points on the same case of the same segment must have the same number of states (i.e. last-state(ABi) = last-state(ABo), last-state(+i) = last-state(+o), and last-state(ARi) = last-state(ARo)). These post-conditions are not usually made explicit in behavioral descriptions. The temporal environments associated with AB must have one more state than those associated with AR. This is true because temporal environments referring to AB apply to every occurrence of AB, which includes all of the occurrences of AR, while temporal environments referring to AR do not refer to the outermost occurrence of AB. A post-condition to this effect will also be implicit rather than explicit. AB is either executed the same number of times as + or one more time. Which one is the case depends on what causes the augmentation to stop execution. Section IV.2.1.5 discusses how the relationship between last-state(body) and last-state(augmentation function) can be determined in a particular program. The pre-condition in the example "last-state(TZ) = last-state(TX)+1" specifies that when the augmentation is used, something has to be done in order to ensure that AB will be executed one more time than +. Note that it is possible to refer to a temporal output in a pre-condition even though it is not possible to refer to an ordinary output. The post-condition $\neg$terminates(A), simply states that like any augmentation, as it stands, segment A cannot terminate execution. This is true because the control flow AR$\rightarrow$AB requires that AB cannot terminate until after it terminates.

Pre-conditions of a temporal input are determined by bringing up any pre-conditions for the corresponding ordinary input which are not internally satisfied. Two changes are made in the logical expression. First, substitution is used so that the expression refers only to external quantities. Second, it is put in a universal quantifier so that it applies to all of the items in the temporal input. In the example, "REAL(X)" becomes "$\forall i \in$TX(REAL(TX$_i$))". This requires that for every state of the temporal environment associated with TX, the value of TX must be a real number. Every pre-condition of the augmentation function must be either internally satisfied, or moved out to the behavioral description of the resulting segment. Occasionally, a temporal output must be included in the behavioral description of the outer segment for the sole purpose of making it possible to bring out an unsatisfied pre-condition.

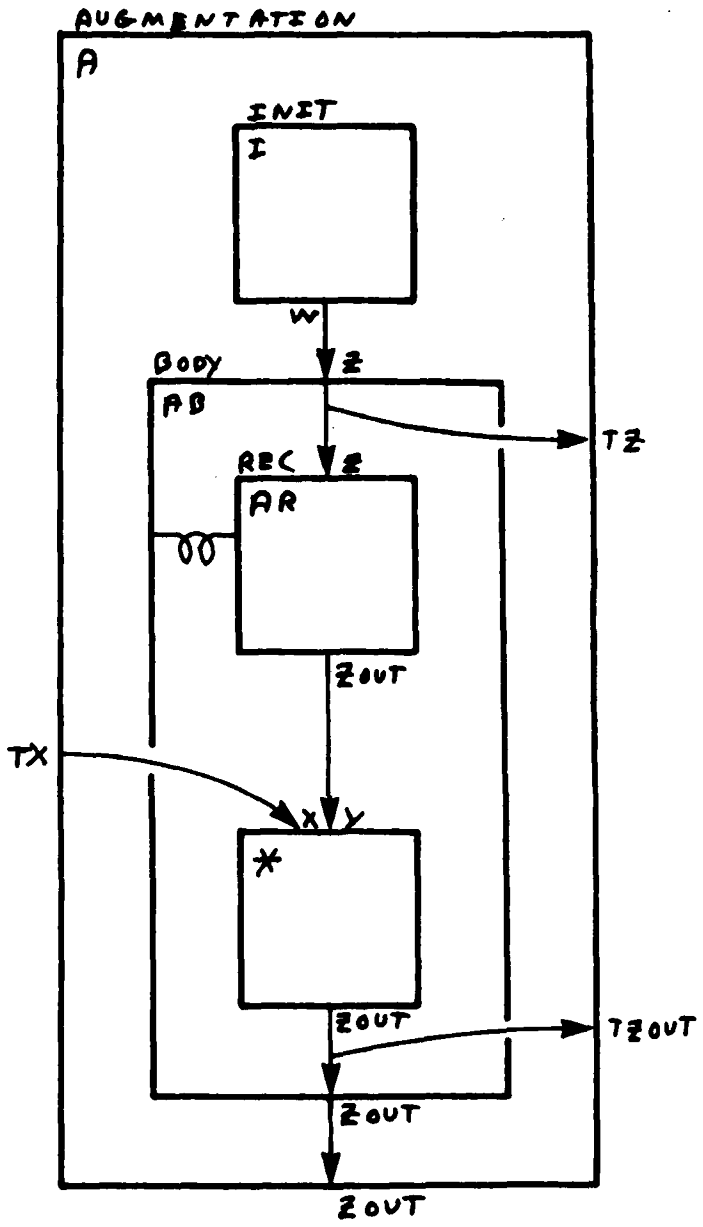
Post-conditions for temporal outputs are copied from post-conditions of the initialization and the augmentation function. As they are moved out they are converted in the same way that the pre-conditions are. For example, the post-condition "Z=X+Y" of + becomes "$\forall i(2 \leq i \leq$last-state(TZ) $\rightarrow$ $TZ_i=TZ_{i-1}+TX_{i-1}$)". The fact that this post-condition applies to all but the first value of TZ follows from the data flow in the augmentation. The post-condition "Z=0" of the initialization I leads to the post-condition "$TZ_1=0$". The post-conditions about the type of Z are moved up in a similar manner. In general an initialization can provide initial values for the augmentation function (as in the example), and it can provide values which are used in each cycle of the loop, but which do not need to be recalculated each time (the example in Figure IV-13 shows the kind of data flow which is used in order to carry a value to every cycle of the loop). The above post-conditions for the temporal output TZ are accurate but not very satisfying. Fortunately, they correspond to a recurrence relation with a known solution. As a result it can be claimed that "$\forall i \in$F($TZ_i= \sum_{j=1,i-1} TX_j$)".

The method described above always leads to recurrence relations for the temporal outputs of an augmentation. Unfortunately, there is no guarantee that they will have nice solutions. However, as shown in the experiment discussed in Chapter V, a large percentage of augmentations have temporal outputs described by simple solvable recurrence relations. The five most common augmentations seen in that experiment were: a sum (which has the augmentation function "Z=Z+_" as in the example), a product (which has the augmentation function "Z=Z*_"), a count (which has the augmentation function "Z=Z+1"), a maximum (which has the augmentation function "IF _>Z THEN Z=_"), and a minimum (which has the augmentation function "IF _<Z THEN Z=_").

Now consider how the post-conditions about the output ZOUT of A are derived. A fundamental problem with tail recursive augmentations, is that there is no simple way to specify that they produce an ordinary output. This is due to the fact that as written they cannot terminate, and therefore cannot produce useful ordinary outputs. The approach taken in the example, is to require, through a pre-condition, that if the augmentation is used, something must be done to make it terminate, and create the data flow path needed to produce the output.

The intention is to specify that ZOUT will be the last value of TZ_i. The data flows AR.ZOUT→AB.ZOUT, and AB.ZOUT→A.ZOUT can carry this value to the outer segment from the innermost invocation of AB. However, it is not possible to put in data flow which shows how ZOUT gets the correct value to start with. (Note that in the code for the loop, this is done by continually setting the variable Z so that it will have the right value in the end. Unfortunately, this simple device cannot be used in the recursive representation of the loop used in the plan.) The third pre-condition of A "$TZOUT_{last\text{-}state(TZOUT)} = TZ_{last\text{-}state(TZ)}$" is used to make the required specification. (This is similar to the pre-condition *"last-state(TZ) = last-state(TX)+1"* in that whatever uses the augmentation must modify it in order to ensure that the pre-condition is true.) Notice that the temporal input TZOUT has been included in the behavioral description solely so that this pre-condition could be formulated.

In order to understand the pre-condition about TZOUT, it is important to note that, in order to make it convenient when combining augmentations together, the states of temporal ports referring to positions executed after the recurrence are numbered so that they correspond with the states of temporal ports referring to positions executed before the recurrence. For example, the tenth occurrence of ABo corresponds to the same execution of the segment AB as the tenth occurrence of ABi. This causes the states in temporal ports after the recurrence (such as TZOUT) to be numbered in reverse temporal order. In the example, $TZOUT_1$ is the value in the last state in order of time while $TZOUT_{last\text{-}state(TZOUT)}$ is the value in the first state in order of time. As a result, the pre-condition requires that the first (in time) value of TZOUT be the same as the last value of TZ. The post-conditions about TZOUT all follow from the pre-condition, and the fact that the data flow preserves the initial value of TZOUT unchanged. Justification links in the figure summarize the arguments given above. Note that like the arguments, the justification links are non-local and multi-level.



Flow Diagram for Figure IV-11

```
                    PROCEDURE A();
                       RETURN(A()*_);
                    END;

          Segment A (the whole program above)
  temporal inputs: TX=(*.X at *i)
   pre-conditions: ∀i∈TX(real(TX_i)), last-state(TZ)=last-state(TX)+1,
                   TZOUT_last-state(TZOUT)=TZ_last-state(TZ)
          outputs: ZOUT
 temporal outputs: TZ=(AB.Z at ABi), TZOUT=(AB.ZOUT at ABo)
  post-conditions: ¬terminates(A), ∀i∈TZOUT(REAL(TZOUT_i)),
                   ∀i∈TZOUT(TZOUT_i=PROD_j=i,last-state(TZOUT)-1 TX_j),
                   REAL(ZOUT), ZOUT=PROD_j=1,last-state(TZOUT)-1 TX_j
        data flow: I.W→AB.Z, AB.ZOUT→A.ZOUT
   justifications: (AB.po1)→A.po2, (*.po2, A.pr3, I.po2)→A.po3,
                   (AB.po1)→A.po4, (A.po3)→A.po5
              pbm: augmentation
            roles: initialization I, body AB, recurrence AR

          Segment I
          outputs: W
  post-conditions: REAL(W), W=1

          Segment AB
           inputs: Z
          outputs: ZOUT
  post-conditions: REAL(ZOUT)
      subsegments: *, AR
        data flow: AB.Z→AR.Z, AR.ZOUT→*.Y, *.ZOUT→AB.ZOUT
   justifications: (*.po1, A.pr3, I.po1)→AB.po1

          Segment *
           inputs: X, Y
   pre-conditions: REAL(X), REAL(Y)
          outputs: ZOUT
  post-conditions: REAL(ZOUT), ZOUT=X*Y
   justifications: (A.pr1)→*.pr1, (AR.po1)→*.pr2

          Segment AR
   recursive link: AB
           inputs: Z
          outputs: ZOUT
  post-conditions: REAL(ZOUT)
```

Figure IV-11: An example of a head recursive augmentation.

The program in Figure IV-11 is a head recursive augmentation which computes the product of the items it receives. The logical structure of this augmentation is very similar to the structure of the augmentation in the prior example. The pre-condition "last-state(TZ) = last-state(TX)+1" has the same meaning as in the last figure. Note that the temporal ports TX and TZOUT have their states numbered in reverse temporal order as discussed above. When post-conditions for the temporal output TZOUT are brought up, effects of the reverse numbering of states becomes apparent. The post-condition "ZOUT=X*Y" of * becomes "$\forall i(1 \leq i \leq \text{Last-State(TZOUT)}-1 \rightarrow TZOUT_i = TZOUT_{i+1} * TX_i)$". The value of $TZOUT_i$ depends on higher numbered values of TZOUT rather than lower numbered values as it would if the augmentation function was before the recurrence.

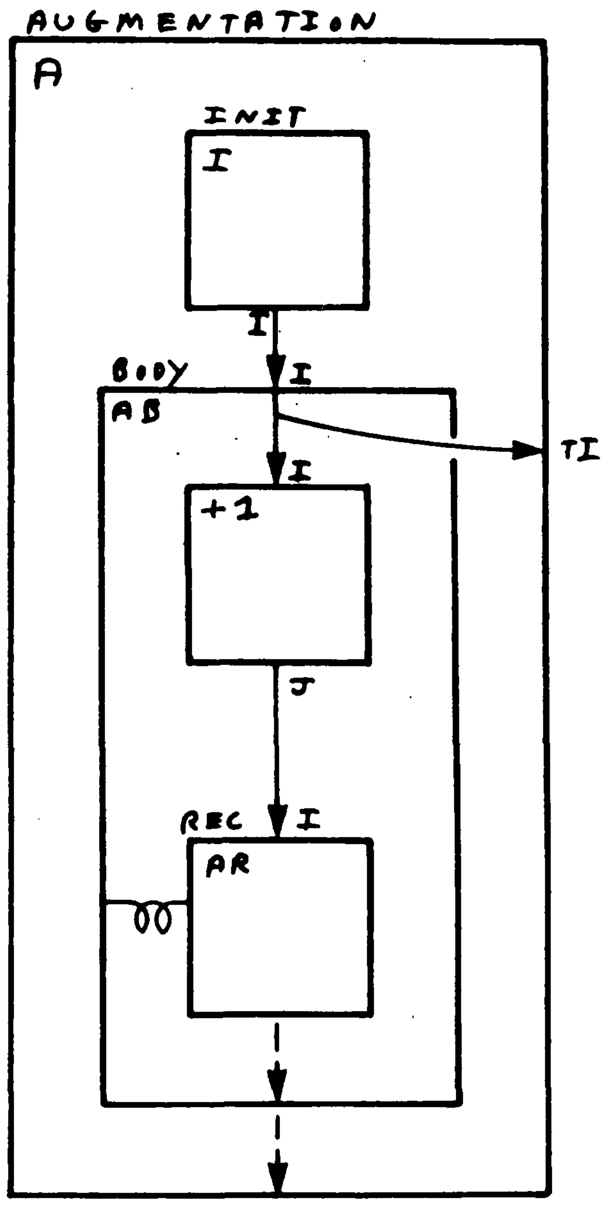
In a tail recursive augmentation, the initialization can provide initial values for the augmentation function (in the example above the initialization provided the initial value $TZ_1=0$). Unfortunately, this is not the case in a tail recursive augmentation. There is no way to connect up data flow in order to link the initialization only to the first value of TY. This is analogous to the problem of specifying the output TZOUT in the example above and is handled in exactly the same way. It is interesting to note that the output of the last value is no problem in a head recursive augmentation. This is done by ZOUT in this example. In order to specify what is needed, the temporal output TZ is introduced so that "$TZOUT_{last\text{-}state(TZOUT)} = TZ_{last\text{-}state(TZ)}$" can be made a pre-condition of the plan. This pre-condition implies that if the augmentation is going to be used, then something has to be done so that the initial value of TZOUT is set to 1.

Given that $TZOUT_{last\text{-}state(TZOUT)}=1$, the recurrence relation for TZOUT can be solved to yield "$\forall i \in TZOUT(TZOUT_i = PROD_{j=i,last\text{-}state(TZOUT)-1}\ TX_j)$". This expression takes advantage of the fact that multiplication is associative and reverses the apparent order of the multiplications. The augmentation in the example has an ordinary output ZOUT. Its value is equivalent to $TZOUT_1$ and it gets its post-conditions from the post-conditions of TZ.

A solution of a recurrence relation is verified by an inductive proof. In the last example, the basis step of this proof is based on the value of TZ_1. Here, however, the basis step is based on $TZOUT_{last\text{-}state(TZOUT)}$. As a result, the proof is only valid if the augmentation terminates so that there is a $TZOUT_{last\text{-}state(TZOUT)}$. In the prior example, there did not have to be any assumption about the termination of the augmentation. In fact tail recursive augmentations are occasionally used in situations where they are not expected to terminate. For example, LISP's read eval print loop. The behavioral description developed for the last example is still valid in that situation. In contrast, it is an implicit pre-condition of a head recursive augmentation that something has to be done to cause it to terminate before it can be used. A head recursion does not begin to compute anything until after the innermost invocation of the body terminates. The post-condition ¬terminates(A) in the plan simply states that as it stands now, the augmentation will not terminate.

A common reason for using a tail recursive augmentation is that the augmentation function is not associative, and a reversed order of exec 'tion is desired. For example, consider an augmentation function of the form "Z=CONS(_, Z)". If this augmentation function is put before the recurrence, then the item TX_1 will be the first item CONSed on and therefore will be the last element in the resulting list. On the other hand, if it is placed after the recurrence, then TX_1 will be the last item CONSed on and therefore will be the first element of the resulting list.

```
                 I = 1;
           LOOP: I = I+1;
                 GOTO LOOP;

         Segment A (the whole program above)
temporal outputs: TI=(AB.I at ABi)
 post-conditions: ¬terminates(A), ∀i∈TI(INTEGER(TIi)), ∀i∈TI(TIi=i)
       data flow: I.I→AB.I
    control flow: AB→A
  justifications: (I.po1, +1.po1)→A.po2, (I.po2, +1.po2)→A.po3
             pbm: augmentation
           roles: initialization I, body AB, recurrence AR

         Segment I
         outputs: I
 post-conditions: INTEGER(I), I=1
```



Flow Diagram for Figure IV-12

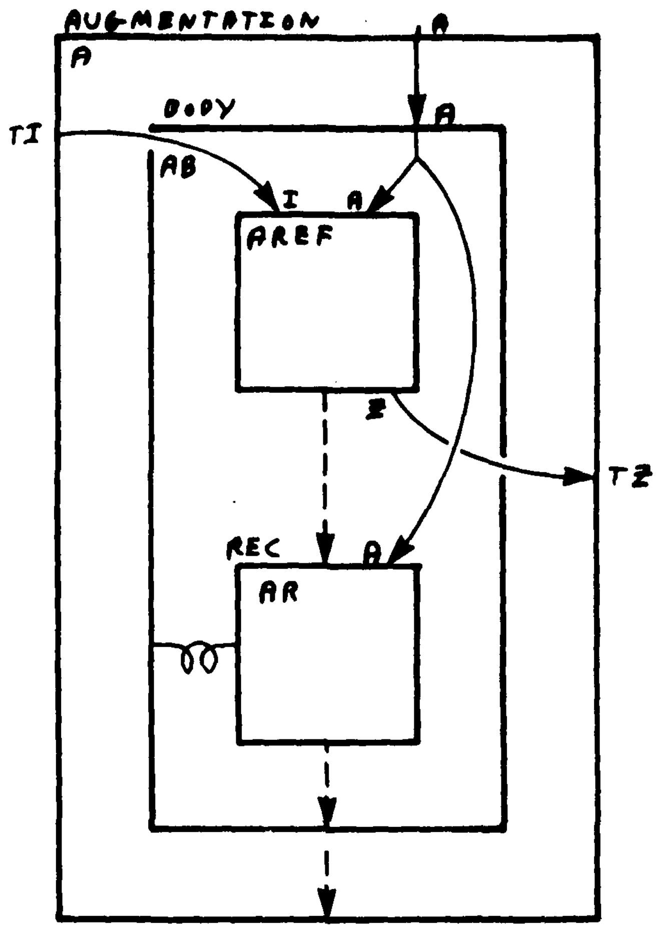
```
          Segment AB
           inputs: I
   pre-conditions: INTEGER(I)
      subsegments: +1, AR
        data flow: AB.I→+1.I, +1.J→AR.I
     control flow: AR→AB
   justifications: (I.pol)→AB.prl
```

```
          Segment +1
           inputs: I
   pre-conditions: INTEGER(I)
          outputs: J
  post-conditions: INTEGER(J), J=I+1
   justifications: (AB.pr1)→+1.pr1

          Segment AR
   recursive link: AB
           inputs: I
   pre-conditions: INTEGER(I)
   justifications: (+1.po1)→AR.pr1
```

Figure IV-12: An example of an augmentation without temporal inputs.

Figure IV-12 shows an example of an augmentation which counts up by 1. The basic post-conditions which are produced for the temporal output TI are "$TI_1=1$" and "$\forall i(2\leq i\leq last\text{-}state(TI)\rightarrow TI_i=TI_{i-1}+1)$". These can be solved to yield "$\forall i\in TI(TI_i=\sum_{j=1,i}1)$", or more compactly "$\forall i\in TI(TI_i=i)$". The primary thing to notice about this augmentation is that it does not have any temporal inputs, and can actually be executed as it stands (though it will not terminate). As will be seen in Section IV.2.1.5 tail recursive augmentations without any temporal inputs are important because they can serve as bases upon which to build complex recursive programs.

```
            LOOP: A(_);
                  GOTO LOOP;

            Segment A (the whole program above)
             inputs: A
    temporal inputs: TI=(AREF.I at AREFi)
     pre-conditions: REAL(A), VECTOR(A), ∀i∈TI(INTEGER(TI_i)), ∀i∈TI(1≤TI_i≤SIZE(A))
   temporal outputs: TZ=(AREF.Z at AREFo)
    post-conditions: ¬terminates(A), ∀i∈TZ(REAL(TZ_i)), ∀i∈TZ(TZ_i=A(TI_i))
          data flow: A.A→AB.A
       control flow: AB→A
     justifications: (AREF.po1)→A.po2, (AREF.po2)→A.po3
                pbm: augmentation
              roles: body AB, recurrence AR

            Segment AB
             inputs: A
     pre-conditions: REAL(A), VECTOR(A),
        subsegments: AREF, AR
          data flow: AB.A→AREF.A, AB.A→AR.A
       control flow: AREF→AR, AR→AB
     justifications: (A.pr1)→AB.pr1, (A.pr2)→AB.pr2

            Segment AREF
             inputs: A, I
     pre-conditions: REAL(A), VECTOR(A), INTEGER(I), 1≤I≤SIZE(A)
            outputs: Z
    post-conditions: REAL(Z), Z=A(I)
     justifications: (AB.pr1)→AREF.pr1, (AB.pr2)→AREF.pr2,
                     (A.pr3)→AREF.pr3, (A.pr4)→AREF.pr4
```



Flow Diagram for Figure IV-13

```
         Segment AR
  recursive link: AB
          inputs: A
  pre-conditions: REAL(A), VECTOR(A),
  justifications: (AB.pr1)→AR.pr1, (AB.pr2)→AR.pr2
```

Figure IV-13: *An example of a conjunction augmentation.*

Figure IV-13 shows an augmentation which takes in a sequence of indices and produces a sequence of items in the vector A. Here, the basic post-condition about the temporal output TZ is just "$\forall i < TZ(TZ_i = A(TI_i))$". The interesting thing about this is that it does not refer to prior values of TZ. This is because the augmentation function does not have data flow to itself on the next iteration as the augmentation functions did in all the prior examples. This is a useful special case because it is particularly easy to understand due to the fact that no recurrence relation needs to be solved. Augmentations like this one which do not have data flow to themselves are called conjunction augmentations. Those which do are called composition augmentations. These names are used in analogy with the conjunction and composition PBMs, discussed above, because these PBMs make a similar distinction based on presence or absence of data flow between segments.

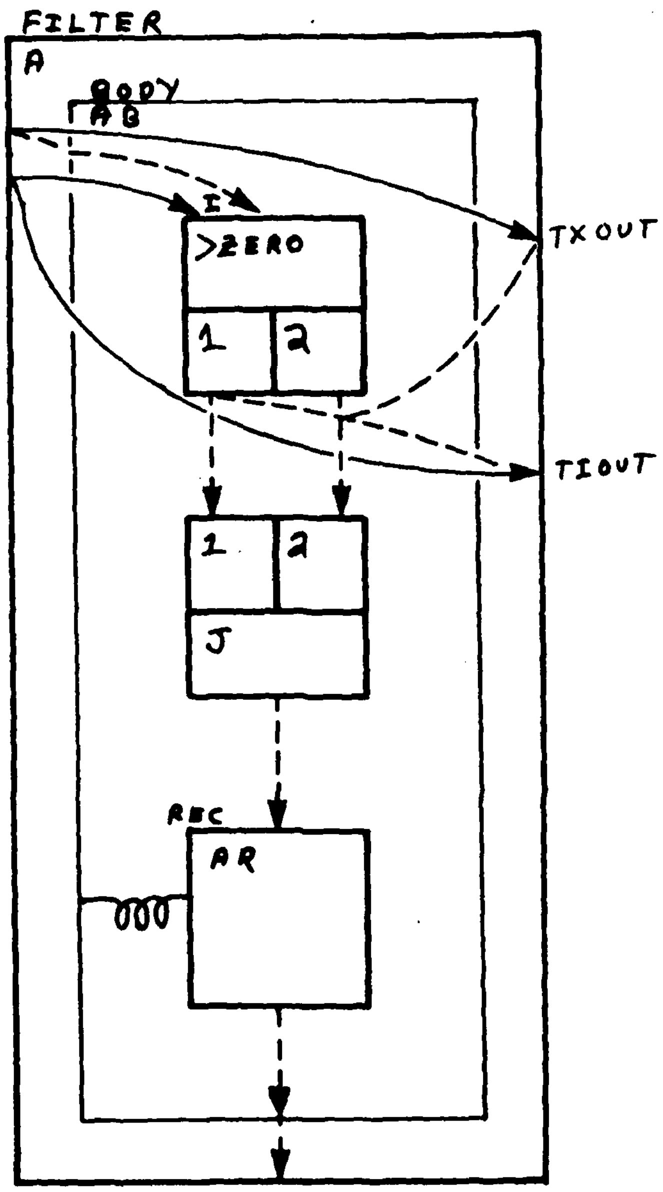
Another thing to notice about this augmentation is that it has an ordinary input A which is an input of the outer segment. This input is passed unchanged to every occurrence of the augmentation function. Pre-conditions involving it have propagated out to the outer segment. Most notable of these is "$\forall i \in TI(1 \leq TI_i \leq SIZE(A))$".

IV.2.1.3 The PBM Filter

The filter PBM is the same as the single self recursion PBM except for some additional restrictions on the body. The body must have only one case. It must have exactly three subsegments. One subsegment is the recurrence, the second must be a split segment which is referred to as the "filter test", and the third must be a join with the same number of cases as the split. The filter test must be constrained to be either always executed before the recurrence, or always executed after the recurrence. There must be a control flow arc from each case of the split going directly to the corresponding case of the join. As a result of this, the filter test and the join look like a conditional with no actions. It is not possible for the filter test to cause the filter to be a split or to terminate. What it does do is create zones of restricted execution within the filter. There can be temporal outputs referring to the outside of the filter test, but not to its inside. There can be temporal inputs referring only to the inputs of the filter test.

Like an augmentation, filters are not useful in isolation. They are only useful as a fragment which can be combined together with augmentations by the PBM temporal composition. Filters can be identified by checking to see whether something analyzed as a single self recursion satisfies the additional requirements associated with a filter. Filters are distinguished from augmentations by the fact that a filter test is a split while an augmentation function must be a straight-line segment.

```
              LOOP: IF _>0 THEN; ELSE;
                    GOTO LOOP;

           Segment A (the whole program above)
   temporal inputs: TI=(>ZERO.I at >ZEROi), TX=(at >ZEROi)
    pre-conditions: ∀i∈TI(INTEGER(TI_i))
  temporal outputs: TIOUT=(TI_i at >ZERO1o), TXOUT=(TX_i at >ZERO1o)
   post-conditions: TIOUT=sub-sequence(TI_i| TI_i>0),
                    TXOUT=sub-sequence(TX_i| TI_i>0),
                    last-state(TIOUT)=number-of(TI_i| TI_i>0), ¬terminates(A),
                    ∀i∈TIOUT(INTEGER(TIOUT_i)), ∀i∈TIOUT(TIOUT_i>0)
    justifications: (>ZERO1.c1)→A.po1, (>ZERO1.c1)→A.po2, (A.po1)→A.po3,
                    (A.pr1, A.po1)→A.po5, (A.po1)→A.po6
               pbm: filter
             roles: body AB, recurrence AR
      control flow: AB→A

           Segment AB
       subsegments: >ZERO, J, AR
      control flow: >ZERO1→J1, >ZERO2→J2, J→AR, AR→AB
```



Flow Diagram for Figure IV-14

```
         Segment >ZERO
          inputs: I
  pre-conditions: INTEGER(I)
  justifications: (A.pr1)→>ZERO.pr1
      CASE1
     conditions: I>0
      CASE2
     conditions: I≤0

         Segment J
      CASE1
      CASE2
```

```
Segment AR
recursive link: AB
```

Figure IV-14: An example of a filter.

The filter in Figure IV-14 takes in a sequence of values and selects out those which are greater than zero. The temporal input TI is activated every time the filter test (>ZERO) is activated. The temporal output TIOUT is activated only when case 1 of >ZERO is activated; i.e. when $TI_i>0$. The temporal output TIOUT is formed from just those values of TI which are greater than zero. The post-condition "TIOUT=sub-sequence(TI_i| $TI_i>0$)" signifies that the sequence of values corresponding to TIOUT is just the same as the values corresponding to TI except that all (and only) the values which satisfy the predicate ($TI_i>0$) are retained while the other values are deleted. The retained elements stay in their original order. All of the other post-conditions about TIOUT can be derived from this one. For example, "last-state(TIOUT) = number-of(TI_i| $TI_i>0$)" signifies that the number of states in TI is the same as the number of elements in TI which are greater than zero.

The temporal ports TX and TXOUT are interesting because they do not refer to anything that is used in the filter itself. The specification "TX=(at >ZEROi)" implies that there must be a value of TX available at each occurrence of >ZEROi, but that that value will not be used anywhere inside the filter. This kind of restricted temporal input is allowed even though it is not actually referring to an input of the filter test because it is associated with the input side of the filter test. The specification "TXOUT=(TX_i at >ZERO2o)" indicates that the temporal output TXOUT is formed from those elements of TX which are seen at the output side of case 2 of the filter test. This is spelled out in the post-condition "TXOUT=sub-sequence(TX_i| $TI_i>0$)". Note that items in TX are being selected based on properties of the corresponding elements in TI, and not based on any properties of the elements of TX themselves. Justification links in the figure summarize the arguments given above.

It is easy to determine the behavioral description for a filter. Analogous to an augmentation, any pre-conditions of the filter test which are not internally satisfied are propagated up to the outer segment. The function "sub-sequence" is used to specify a post-condition for each temporal output of the filter. Each temporal output is a sub-sequence of the temporal input it comes from, where the restriction predicate is taken from the conditions of the case of the filter test which the temporal output's temporal environment refers to. The other post-conditions shown in the example follow from the sub-sequence post-conditions.

The description of how the behavioral description is derived, given in the last paragraph, applies to filters, like the one in the example, which are conjunction filters in that the filter test does not have any data flow to itself. Most filters are conjunction filters. However, it is possible for the filter test to have data flow to itself. In that case, the analysis is essentially the same except that it may be necessary to solve a recurrence relation based on the conditions of the filter test in order to determine a convenient restriction predicate to put in the sub-sequence post-condition.

```
LOOP: ...
      IF I>0 THEN DO;
         Y = Y+A(I);
         Z = Z+X;
      END;
      ...
      GOTO LOOP;
```

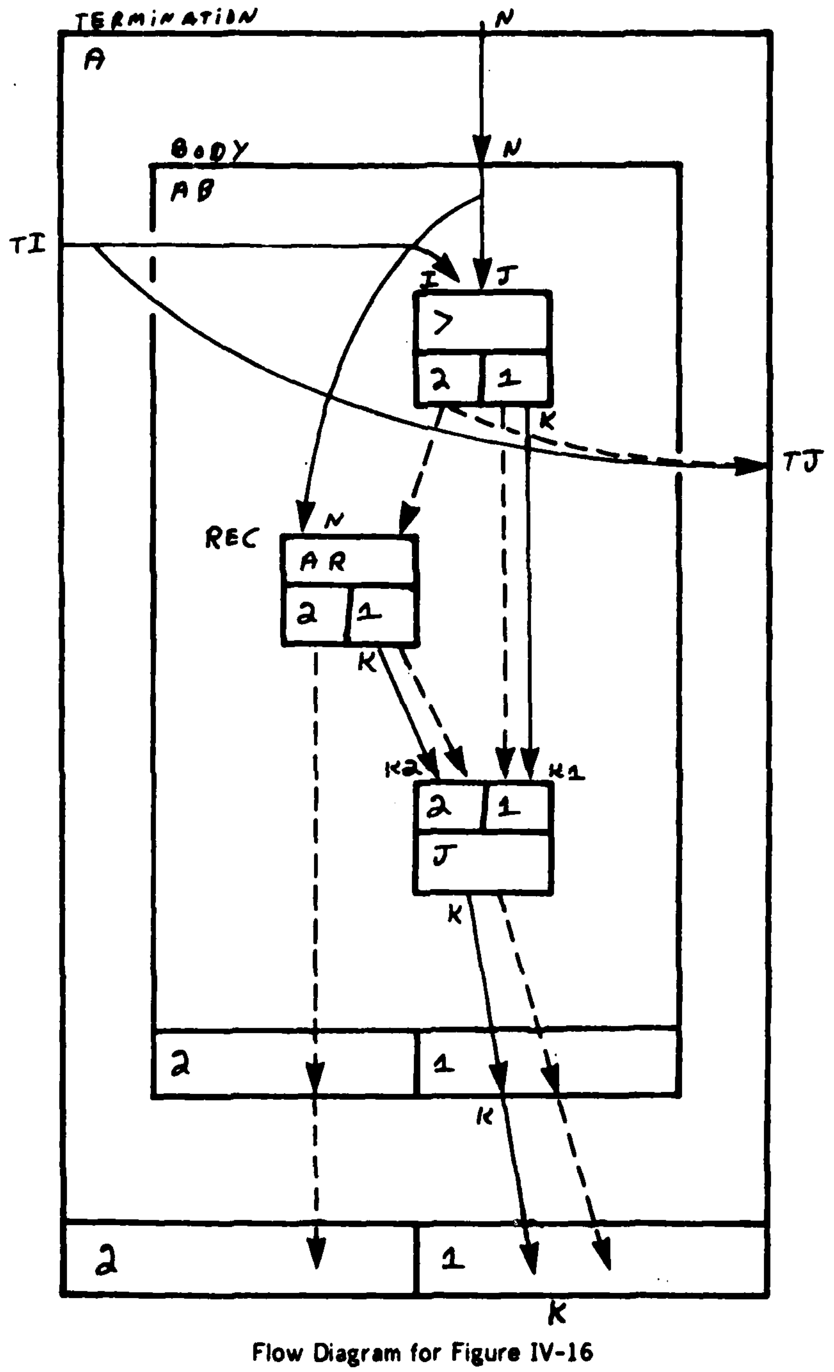
Figure IV-15: An example of computation controlled by a filter.

Filters are used to restrict temporal sequences of values. They correspond to a test in a loop which cannot cause termination, but which create environments which are executed only some of the times the loop as a whole is executed. The loop in Figure IV-15 contains a filter which has the plan shown in Figure IV-14. The loop also has computation corresponding to two summation augmentations in the restricted execution environment created by the filter. The filter selects out the positive elements carried by I. This controls what values of A(I) get added up by controlling what values of I are seen by the summation augmentation. The filter also restricts the values of X available to the augmentation "Z=Z+X". It does this because values of X which do not correspond to positive values of I do not reach the augmentation because the augmentation is not executed. It is an important characteristic of a filter that it restricts every sequence of values used by computations in its execution environment, not just the sequences of values it explicitly tests.

IV.2.1.4 The PBM Termination

The termination PBM is the same as the single self recursion PBM except for some additional restrictions on the body. The body must have two subsegments. One subsegment is the recurrence, and the other must be a split segment which is referred to as the "termination test". Let M be the number of cases in the termination test. The body must also have M cases. There must be a control flow arc from one of the cases of the termination test to the NIL case of the recurrence. Control flow from M-1 cases of the termination test must bypass the recurrence and lead to the output side of a case of the body. There must be a control flow path from each of the M cases of the recurrence to the corresponding case of the body. This requires M-1 joins to join the control flows coming from the recurrence with the control flows coming directly from the termination test. Unlike an augmentation or a filter, a termination can terminate. Temporal outputs can be associated with the body, the recurrence and the exterior of the termination test. Temporal inputs can refer only to the input side of the termination test.

Like augmentations and filters, terminations are fragments and are not useful in isolation. They are combined with augmentations and filters by the temporal composition PBM. Terminations are identified as a special case of single self recursion. They are distinguished from filters due to the fact that there is control flow from the termination test which bypasses the recurrence.



Flow Diagram for Figure IV-16

```
          LOOP: IF _>N THEN GOTO EXIT;
                GOTO LOOP;
          EXIT: ...

          Segment A (the whole program above)
           inputs: N
 temporal inputs: TI=(>.I at >i)
  pre-conditions: INTEGER(N), ∀i∈TI(INTEGER(TI_i))
temporal outputs: TJ=(TI_i at >2o)
 post-conditions: ∀i∈TJ(TJ_i=TI_i), ∀i∈TJ(TJ_i≤N),
       data flow: A.N→AB.N, AB1.K→A1.K
    control flow: AB1→A1, AB2→A2
  justifications: (>2.c1)→A.po2, (>1.c1)→A1.c1, (>1.po1)→A1.po4,
                  (>1.c1, >1.po2)→A1.po5, (>2.c1)→A2.c1
             pbm: termination
           roles: body AB, Recurrence AR
       CASE1
      conditions: ∃i∈TI(TI_i>N)
         outputs: K
 post-conditions: terminates(A), last-state(TI)=MIN({i∈TI| TI_i>N}),
                  last-state(TJ)=last-state(TI)-1,
                  INTEGER(K), K=first(TI_i| TI_i>N)
       CASE2
      conditions: ¬∃i∈TI(TI_i>N)
 post-conditions: ¬terminates(A), last-state(TI)=last-state(TI)

          Segment AB
           inputs: N
  pre-conditions: INTEGER(N)
     subsegments: >, AR, J
       data flow: AB.N→>.J, AB.N→AR.N, >1.K→J1.K1, AR1.K→J2.K2, J.K→AB1.K
    control flow: >2→AR, >1→J1, AR1→J2, J→AB1, AR2→AB2
  justifications: (A.pr1)→AB.pr1
       CASE1
         outputs: K
       CASE2

          Segment >
           inputs: I, J
  pre-conditions: INTEGER(I), INTEGER(J)
  justifications: (A.pr2)→>.pr1, (AB.pr1)→>.pr2
       CASE1
      conditions: I>J
         outputs: K
 post-conditions: INTEGER(K), K=I
       CASE2
      conditions: I≤J

          Segment AR
  recursive link: AB
           inputs: N
  pre-conditions: INTEGER(N)
  justifications: (AB.pr1)→AR.pr1
       CASE1
         outputs: K
       CASE2
```

```
         Segment J
         outputs: K
     CASE1
          inputs: K1
 post-conditions: K=K1
     CASE2
          inputs: K2
 post-conditions: K=K2
```

Figure IV-16: An example of a termination.

The termination in Figure IV-16 takes in a sequence of values and terminates as soon as it sees an item greater than N, returning that item. This is an extension of the plan in Figure III-8. The discussion of that figure did not discuss how the behavioral description of the outer segment can be determined.

In order to understand a termination, it must be looked at from two different points of view. On one hand, the termination can cause termination of a recursive program. This has implications reaching throughout the program. On the other hand, the termination acts just like a filter. It takes in sequences of values and outputs restricted sequences of values. Terminations restrict sequences of values by truncating them. They pass through all of the values up to a point, and then they do not pass through any more at all. Like filters, terminations usually truncate many sequences of values in addition to the ones they explicitly test.

Consider how the behavioral description in the example can be derived. The pre-conditions of the termination test (>) are propagated up to the outer segment (A). The temporal output TJ is associated with the output side of case 2 of >. It is activated every time > is activated until case 1 of > is activated. When this happens, the termination terminates and neither > nor TJ is ever activated again. TJ outputs those elements of TI which are seen at the output of case 2 of >. The first and second post-conditions of A state that each item in TJ is the same as the corresponding item in TI, and that each of these values must have triggered case 2 of >.

The termination has two cases which reflect the fact that it either terminates, or it does not. If there is an item in TI which is greater than N then case 1 of > will be activated, activating case 1 of the outer segment A. When this is the case, the termination terminates. The second post-condition of case 1 of A "$\text{last-state}(TI) = MIN(\{i<TI| TI_i>N\})$" reflects the fact that A terminates on the first TI_i which is greater than N. The post-condition "$\text{last-state}(TJ) = \text{last-state}(TI)-1$" reflects the fact that when the termination terminates, TJ is activated for every state of TI except the last one.

Note that this specifies how many states are in the temporal input TJ. This reflects the fact that the PBM temporal composition adds a termination into a recursive program in such a way that it processes the items in its temporal inputs one at a time, and in such a way that the termination prevents any more items in the inputs from being produced after the program terminates. As a result, unlike an augmentation or a filter, a termination controls the number of states in its temporal inputs.

In case 1, A has an ordinary output K which corresponds to the value of TI which caused termination. The post-condition "$K=\text{first}(TI_i| TI_i>N)$" asserts that this value is the first value which is capable of causing termination. A termination can have three basic kinds of outputs. It can have an ordinary output corresponding to the item which causes termination (K in the example). Searches tend to have this kind of output. It can have a temporal output which corresponds to all of the items which fail the termination test (TJ in the example). It can also have a temporal output which corresponds to all of the items tested, up to and including the one which causes termination. Such a temporal output is specified by referring to the input side of the termination test.

Case 2 of A describes the situation where the termination does not terminate. In that case, TJ is the same length as TI. Every item in the input appears in the output. The justification links in the figure summarize the arguments above.

The behavioral description of an arbitrary termination is developed following the pattern above. Any pre-conditions of the termination test which are not internally satisfied are moved up to the outer segment. The conditions of the cases of the termination test are used to derive the conditions of the cases of the outer segment, and the clauses specifying the number of states in the temporal ports. The output temporal sequences associated with the non-terminating case of the termination test are asserted to be truncations of the input temporal sequences. It is possible, but unusual, for a termination test to have data flow to itself. If it does, then the process of deriving the behavioral description becomes more difficult in ways analogous to what happens with the other PBMs above.

IV.2.1.5 The PBM Temporal Composition

The PBM temporal composition builds a complex single self recursive program out of simpler ones. It has five types of roles. It can have an optional role named the basic recursion. It must be possible to build up the segment which fills this role by means of the PBM single self recursion. It is also assumed that the control flow is completely specified in the basic recursion. It can have zero or more roles named augmentation1, augmentation2, etc. It must be possible to build up the segments which fill these roles by means of the PBM augmentation. It can have zero or more roles named filter1, filter2, etc. It must be possible to build up the segments which fill these roles by means of the PBM filter. It can have zero or more roles named termination1, termination2, etc. It must be possible to build up the segments which fill these roles by means of the PBM termination. It can have zero or more joins named join1, join2, etc. In a given situation, the PBM must have at least one role which is not a join. Note that it can already be seen that the PBM temporal composition does not treat its subsegments like black boxes because it makes explicit restrictions on their internal structure.

The subsegments are connected together by acyclic data flow and control flow. The only ordinary data flow allowed is between ordinary ports of the outer segment, and ordinary ports of the subsegments. There cannot be a data flow from an ordinary port of one subsegment to an ordinary port of another subsegment. A control flow arc must originate on each terminating case of a subsegment. A terminating case is one which has a post-condition "terminates(subsegment)" in it. Only terminations have any terminating cases. These control flows connect up to the output sides of the cases of the resulting segment. There can be joins joining together this control flow in order to collapse some of the cases of the resulting segment.

In order to completely specify the way the subsegments are combined, an additional device is introduced. This device, called "temporal data flow", connects temporal outputs to temporal inputs. A temporal data flow from a temporal output (TE) to a temporal input (TF) signifies two things. First, it specifies that the data values available at TE will be transmitted to TF (i.e. $TF_i=TE_i$). Second, it specifies that the execution environments associated with the two ports will operate in synchrony so that the Ith state of TE will occur at the same time as the Ith state of TF. This implies that TE and TF must have the same number of states. There is no directionality implied in this second aspect of temporal data flow. Any information discovered about the number of states in one temporal port can be applied to the other. It is possible for a temporal data flow to be degenerate in that it signifies only that two execution environments are identified, and does not correspond to any transfer of data.

A temporal data flow from TE to TF is only allowed if it is possible to synchronize the execution of the two temporal environments. For example, a temporal data flow is not allowed if the execution environment of the source is after the recurrence in a subsegment, and the execution environment of the destination is before the recurrence in a subsegment, or vice versa. Note that the only temporal inputs which are allowed are those which refer to the inputs of an augmentation function, a filter test, or a termination test. In order to be well formed, an instance of the PBM temporal composition must have exactly one temporal data flow terminating on each temporal input of a subsegment.

The temporal data flow is required to be acyclic. As a result, there must be one or more subsegments which do not have any temporal data flow terminating on them. These subsegments are referred to as "root" subsegments. The basic recursion, if there is one, must be a root subsegment. A subsegment (A) is said to be "closer to a root" than another subsegment (B) if there is a path of temporal subsegment data flows starting on A and ending on B. Every termination subsegment must be ordered with respect to every other subsegment by the relation closer to the root.

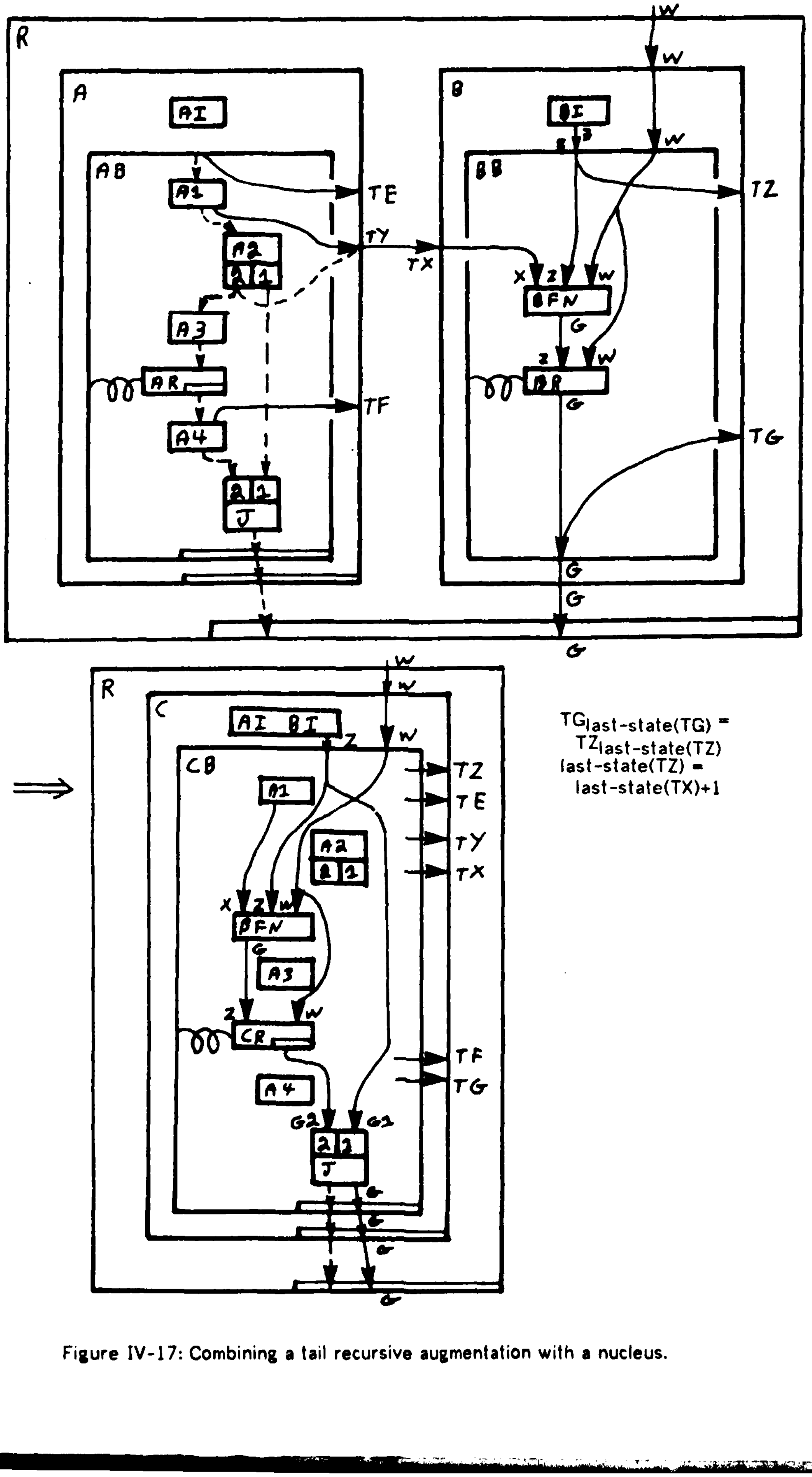
The PBM temporal composition is intimately tied up with the idea of temporal abstraction. One way to see the relationship between the two is to consider how a plan utilizing the PBM temporal composition can be converted into a plan using the PBM single self recursion which therefore does not use temporal abstraction. This is done by tearing the subsegments apart and recombining the pieces. The process starts by selecting the basic recursion, if there is one, or else any root subsegment as a nucleus. A second subsegment is then selected which does not have temporal data flow from any subsegment other than the nucleus. The nucleus and the second subsegment are then combined to produce a new nucleus.

The process continues until all of the subsegments are combined together to yield the resulting single self recursive program. At each stage of the process, the nucleus is capable of being built up by means of the single self recursion PBM, and therefore can fill the basic recursion role of the PBM temporal composition. Each stage of the process can therefore be represented as an instance of the PBM temporal composition. The process is guaranteed to halt with everything combined into one segment because the temporal data flow is acyclic.

IV.2.1.5.1 Constructing the Result of a Temporal Composition

The process of building up the result of a temporal composition will be discussed in detail in three parts. The three parts correspond to the fact that at each step of the process, the subsegment being combined with the nucleus is either an augmentation, a filter, or a termination. Each of the parts discusses how the combination is performed, and what the result looks like.

The flow diagrams in Figure IV-17 show how a tail recursive augmentation is combined with a nucleus single self recursion. The data flow associated with the nucleus has been omitted in order to make the diagrams clearer. In the first step of combining the two segments, their outer segments (A, B), bodies (AB, BB), and recurrences (AR, BR) are identified with each other. The two initializations (AI, BI) are combined to form the initialization of C. The augmentation function is then placed in the control flow in the body of the nucleus. The source of the temporal data flow from A



$TG_{\text{last-state}(TG)} = TZ_{\text{last-state}(TZ)}$
$\text{last-state}(TZ) = \text{last-state}(TX)+1$

Figure IV-17: Combining a tail recursive augmentation with a nucleus.

to B (A.TY→B.TX) specifies where it will go. There are four possible cases based on the execution environment of the temporal output (which is the output side of case 2 of A2 in the example). If it refers to the output side of a segment in the body of the nucleus (as in the example), then the augmentation function is placed in the control flow immediately after the indicated case of the segment. (The requirement above that the control flow be complete in the basic recursion is needed in order to assure that there is no ambiguity involved in placing the augmentation function.) If the temporal output indicates the input side of a segment in the body, then the augmentation function is placed immediately before the indicated case of the segment. If it refers to the output side of the body itself, then the augmentation function is placed right before the end of the body. If it refers to the input side of the body, then the augmentation function is placed at the very beginning of the body. The last case is the default case if there is no temporal data flow from the nucleus to the augmentation. Note that in the figure, the temporal data flow to TX could have come from TE, but that it could not have come from TF because TF is after the recurrence.

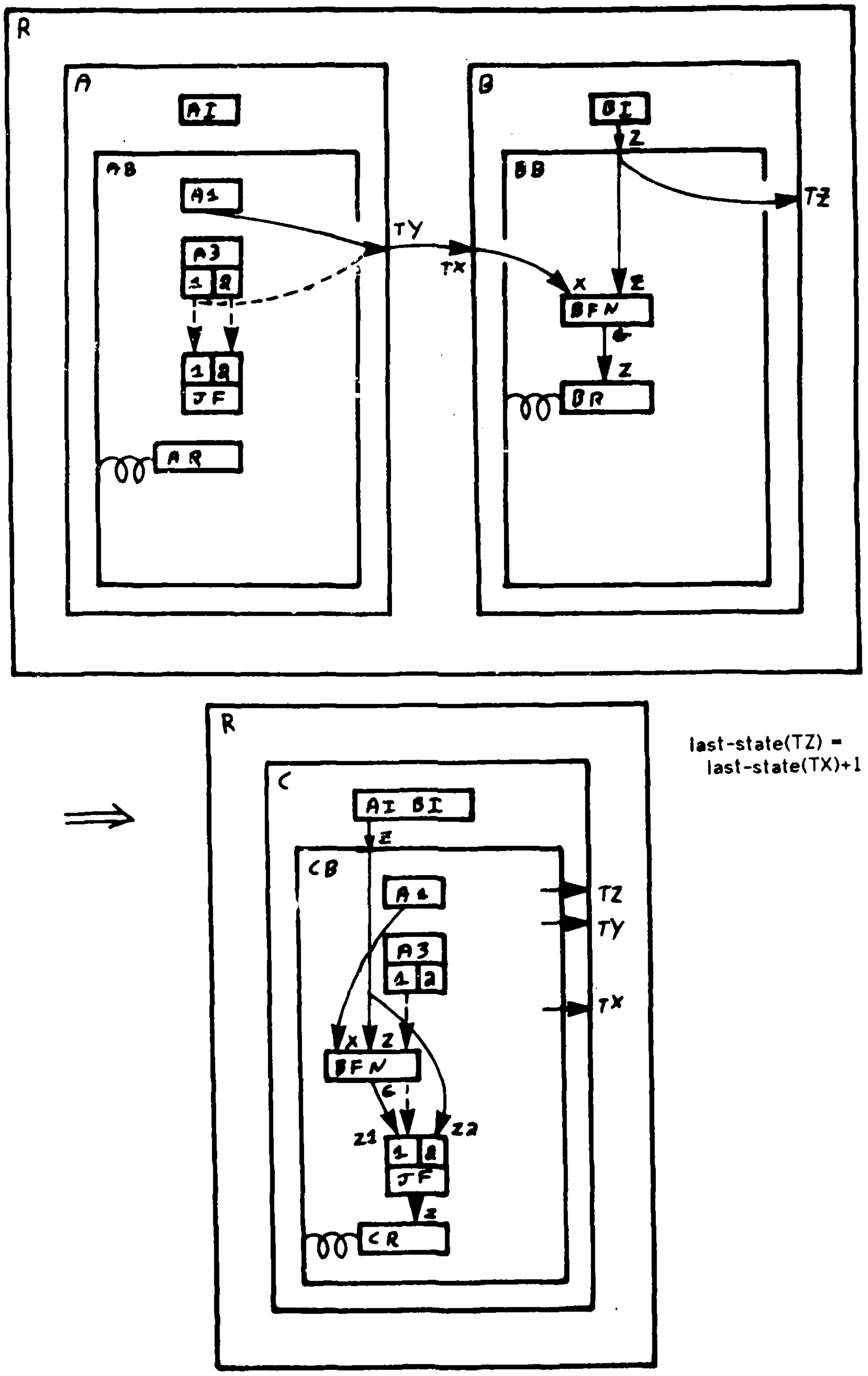
Once the augmentation function is positioned, a data flow is inserted from the internal port associated with the temporal output to the input port associated with the temporal input. If the temporal output is associated with the input side of a subsegment of the body, and refers to an input of that segment, then the data flow is put in coming directly from the source of the ordinary data flow terminating on that input. Analogous steps are taken with respect to temporal outputs associated with the output side of the body. If the temporal data flow is degenerate, then no data flow arc needs to be inserted.

There may be more than one temporal data flow from the nucleus to the augmentation. When this is the case, the augmentation function is placed in one of the indicated positions so that all of the required data flows can be created. If no single position is adequate, then the temporal data flow is ill-formed. The adequate position, if there is one, is the position which is constrained by control flow to be after all of the others.

Once the augmentation function is positioned, all of the data flow in the augmentation is added into the nucleus. This may call for additional ports to be added to the body of the nucleus (such as Z, W, and G in the example). Similarly, all of the temporal ports of the augmentation become temporal ports of the nucleus. TX becomes a temporal output instead of a temporal input because there is now internal data flow satisfying its internal port.

There are two additional complexities. First, the augmentation has an ordinary output G. The augmentation has a pre-condition "$TG_{\text{last-state(TG)}} = TZ_{\text{last-state(TZ)}}$" in order to get the correct value to this output. This pre-condition is satisfied during the process of adding in the augmentation by routing the data flow carrying the output G through the join J, and adding in the data flow CB.Z→J1.G1 which sets up the correct starting value. If the nucleus had not had a termination added into it yet, then this transformation could not be made until later (see the discussion of Figure IV-21).

The other complexity comes from pre-conditions of the form "last-state(TZ) = last-state(TX)+1". In the example, this pre-condition is being satisfied because the augmentation function has been placed after the termination test (A2) in the nucleus. If the temporal data flow had been A.TE→B.TX, then the result would have ensured that "last-state(TZ) = last-state(TX)". As a result, it can be seen that pre-conditions of this type are actually constraints on temporal data flow. If there was no termination test in the nucleus then this pre-condition could not be satisfied until later (see the discussion of Figure IV-21).



last-state(TZ) =
last-state(TX)+1

Figure IV-18: Moving an augmentation into a filter.

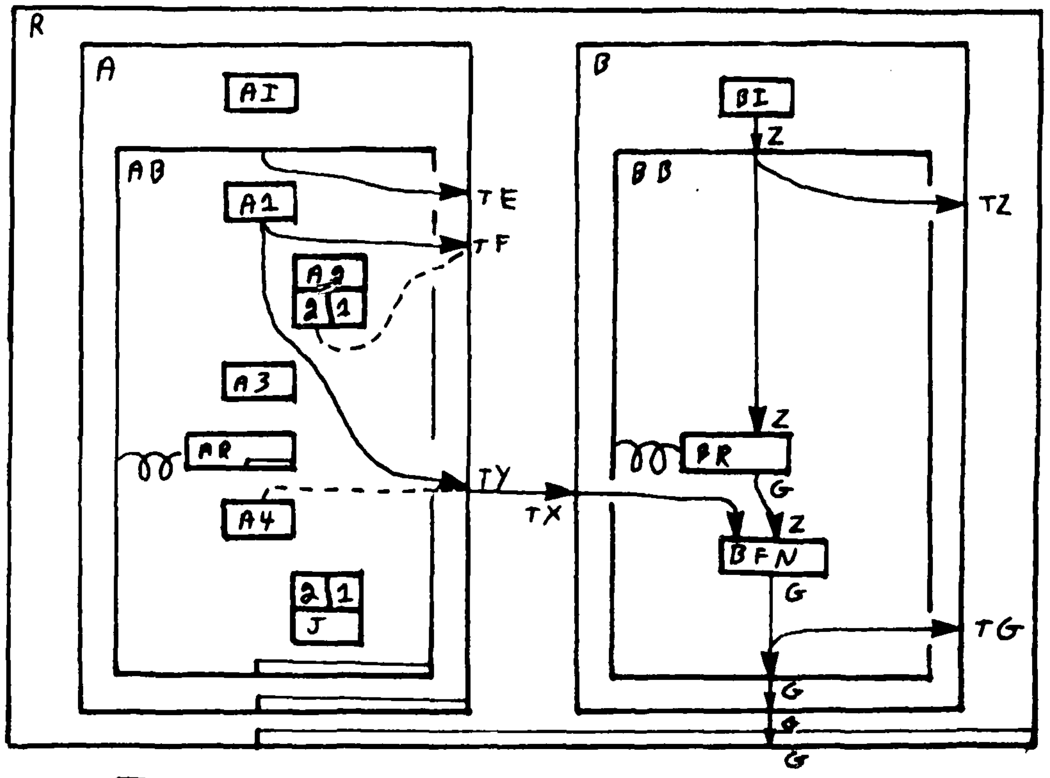
The temporal data flow between the nucleus and the augmentation in Figure IV-18 specifies that the augmentation function BFN is to be put in the execution environment controlled by case 1 of the filter A3. The purpose of the figure is to show that when the augmentation is combined with the nucleus, the data flow from BFN to itself has to be routed through the join JF. This is necessary because BFN is not executed on every iteration of the resulting single self recursion, but rather only when case 1 of A3 is executed. The data flow path from CB.Z through case 2 of JF to CR.Z is needed when BFN is not executed in order to pass through the value Z so that it will be available next time BFN is executed.

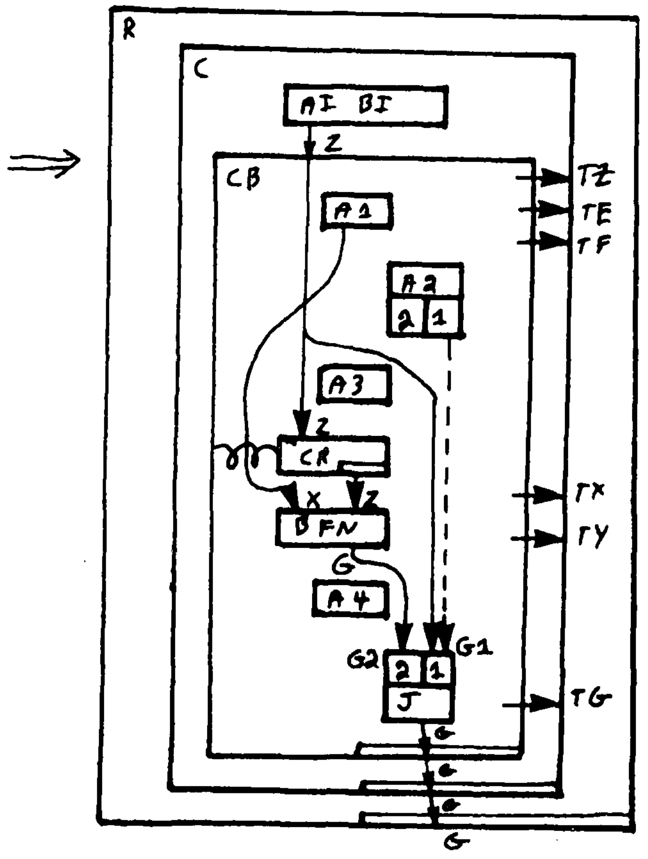
The diagrams in Figure IV-19 show a head recursive augmentation being combined with a nucleus. Here, the augmentation function is placed after the recurrence. The two segments are combined in essentially the same way as in the prior figures, by putting the two plans on top of each other, putting together everything from each of them, with a few additions. This example is given in order to show how a pre-condition (such as "$TG_{last\text{-}state(TG)} = TZ_{last\text{-}state(TZ)}$") which specifies an initial value for a head recursive augmentation function is satisfied. In the example, this is done by routing the data flow carrying G through the join J and adding the data flow CB.Z→J1.G1. This is exactly the same process discussed above for routing the output of a tail recursive augmentation function. An interesting transformation can be applied here. Since the initialization does nothing but provide this initial value, it can be moved to a position on the control flow A21→J1. In this position, it is still only executed once, and it can provide the input for J1.G1. This transformation is advantageous because it eliminates the input Z of the body CB. In this example, the position the augmentation function is placed in implies that "last-state(TG) = last-state(TZ2)+1" due to the fact that the augmentation function precedes the join J. If it was placed after J then "last-state(TG) = last-state(TZ2)" would be true.

Combining a filter and a nucleus is just the same as combining an augmentation and a nucleus (see Figure IV-20). The filter and the join are placed in the result as a unit. The way the temporal data flow A.TY1→B.TX1 appears in the result is interesting. It contributed to the form of the temporal output TZ1 of C. TZ1 refers to the same execution environment as TZ1 of B, and the same internal port as TY1 of A. The temporal port TZ undergoes a similar transformation.

The diagrams in Figure IV-21 show how a termination test (like A2 in the previous examples) gets added into a nucleus. The termination is added in essentially just like an augmentation. Note, however, that a new case gets added to the nucleus and there are some other far-reaching changes. When an augmentation or filter is added into the nucleus, it adds some new computation, but it does not disturb what the nucleus used to do in any way. This is not the case with a termination. The result may terminate sooner than the original nucleus, and therefore compute less. A more indirect way in which a termination can affect a program is that it could be put in between two subsegments of the body which are required by some temporal data flow to be in the same execution environment. This would cause them to no longer be in the same environment. In order to prevent this from happening, a temporal composition is not allowed to have a temporal data flow to a termination which would cause the termination test to be put in a position where some segment before the test would be required to be in the same execution environment as a segment after the test.

The placement of the termination test divides the interior of the body into two sections: the region before the test and the region after the test. The segments which are situated before the test will be executed one more time than those after the test. Therefore, the test in the figure





$TG_{\text{last-state(TG)}} = TZ_{\text{last-state(TZ)}}$

last-state(TZ) = last-state(TX)+1

Figure IV-19: Combining a head recursive augmentation with a nucleus.

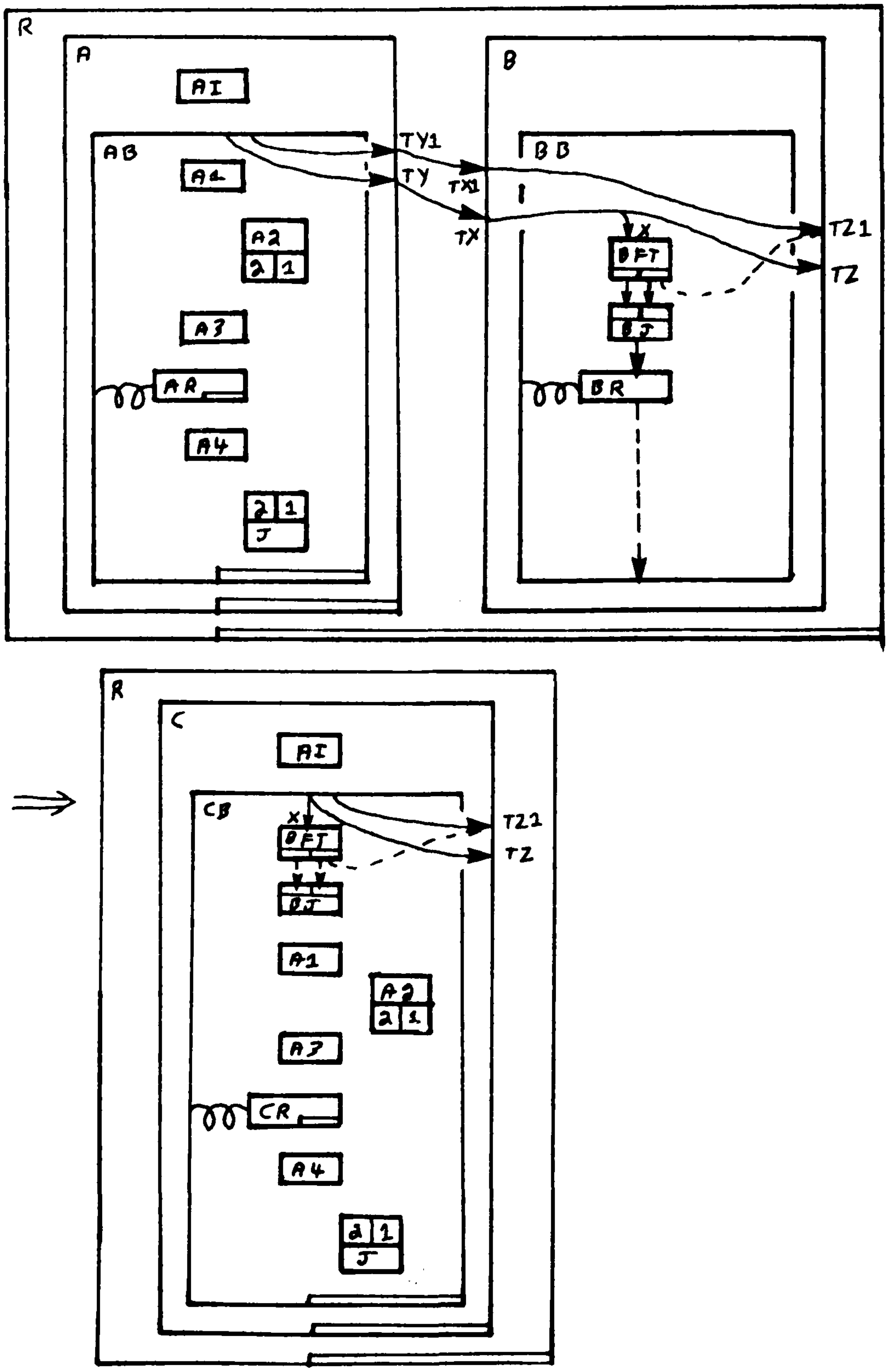
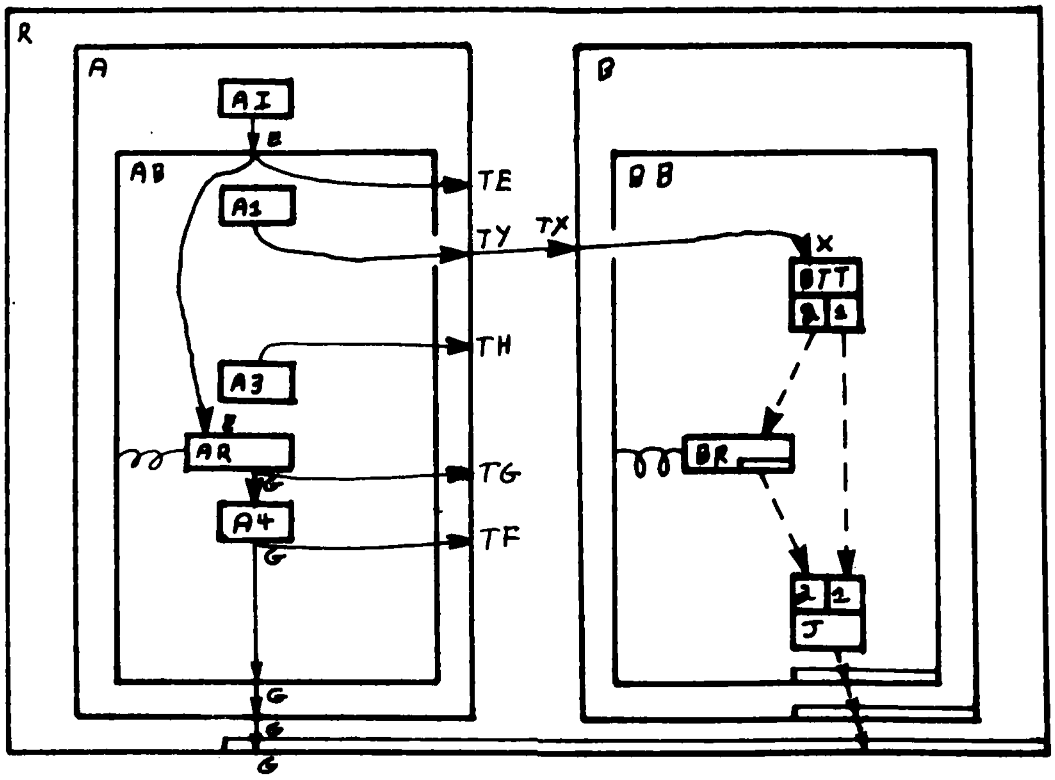
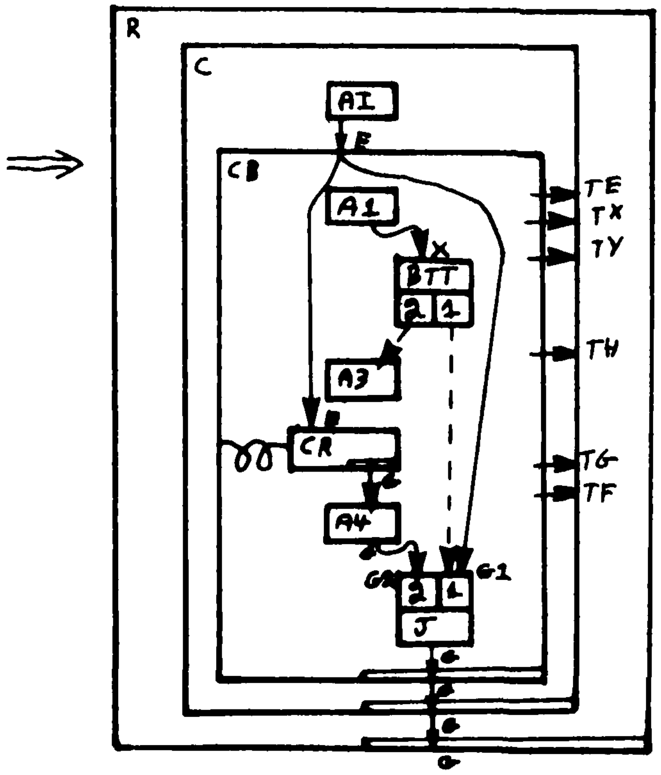


Figure IV-20: Combining a filter with a nucleus.





$TG_{\text{last-state}(TG)} = TE_{\text{last-state}(TE)}$
last-state(TE) = last-state(TY)
last-state(TE) = last-state(TH)+1
last-state(TE) = last-state(TF)+1

Figure IV-21: Combining a termination with a nucleus.

establishes the relationships "last-state(TE) = last-state(TH)+1", and "last-state(TE) = last-state(TY)". The join divides the part of the body below the recurrence into two similar regions. Its placement in the figure is chosen in order to satisfy the pre-condition "last-state(TE) = last-state(TF)+1".

When a termination is added, it must check if there are any unsatisfied pre-conditions of the type "last-state(TE) = last-state(TY)", or of the type "$TG_{\text{last-state(TG)}} = TE_{\text{last-state(TE)}}$". If there are, then it must satisfy them. Four of these pre-conditions are being satisfied in the example.

When a second termination is added into a nucleus (see Figure IV-22) the basic action is to create another case of the body. The only difficulty is associated with the pre-conditions which are satisfied by terminations. In the example, A3 ends up between two termination tests. As a result, neither "last-state(TE) = last-state(TX)+1", nor "last-state(TE) = last-state(TX)" is true, because the relative number of states depends on which test terminates the loop. If a pre-condition along these lines were desired, then it would have to be a more complex expression effectively depending on which way the program terminates.

There is also a complication involving routing outputs of tail recursive augmentation functions to the outside. In the example, the data flow has to be routed through the new join in order to reach both output cases. Similar measures are required in order to make sure that inputs for head recursive augmentation functions are provided correctly. In general, when a second termination is added, it has to check that all of the pre-conditions which terminations satisfy stay satisfied.

The data flow and control flow between the subsegments and the output side of the outer segment of the temporal composition is used to guide the way multiple terminations are added in. In the example, control flow goes from each termination to a separate case of the result. This causes the new termination to create a new case in the nucleus. The control flow could have gone to the same case through a join. This would have caused join J and join BJ to be merged in C so that C had the same number of cases as A. Similarly, it was the fact that there was a data flow A1.G→R2.G which caused the data flow for G to get routed through the new join BJ. If it had not been there, then this would not have happened.

After all of the subsegments of a temporal composition have been combined into the nucleus, the nucleus is taken to be the result. It has cases corresponding to all of the terminating cases of the terminations, and an additional case which corresponds to the possibility of its never terminating. If it can be shown that it will always terminate, then this case can be omitted.

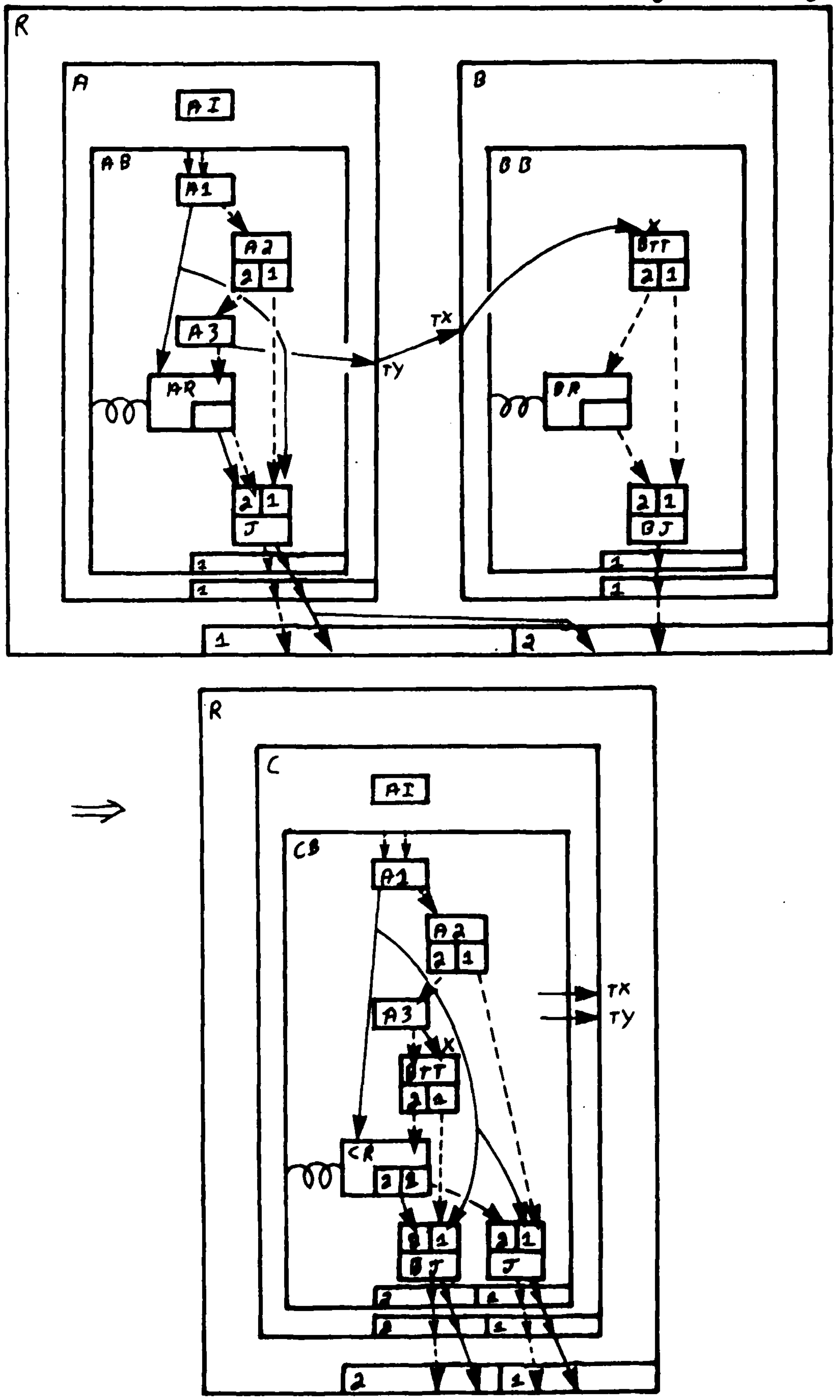
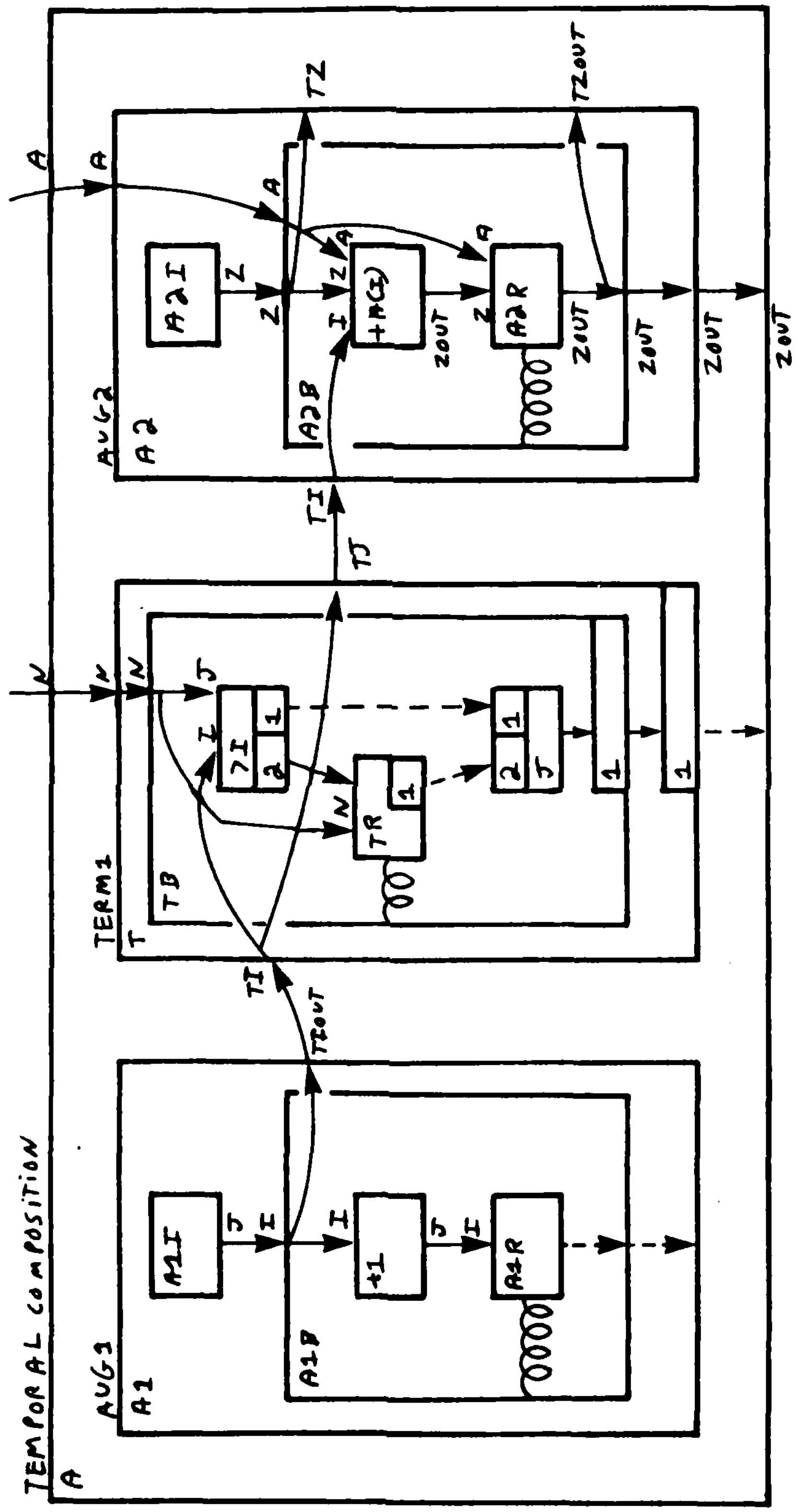


Figure IV-22: Combining a second termination with a nucleus.

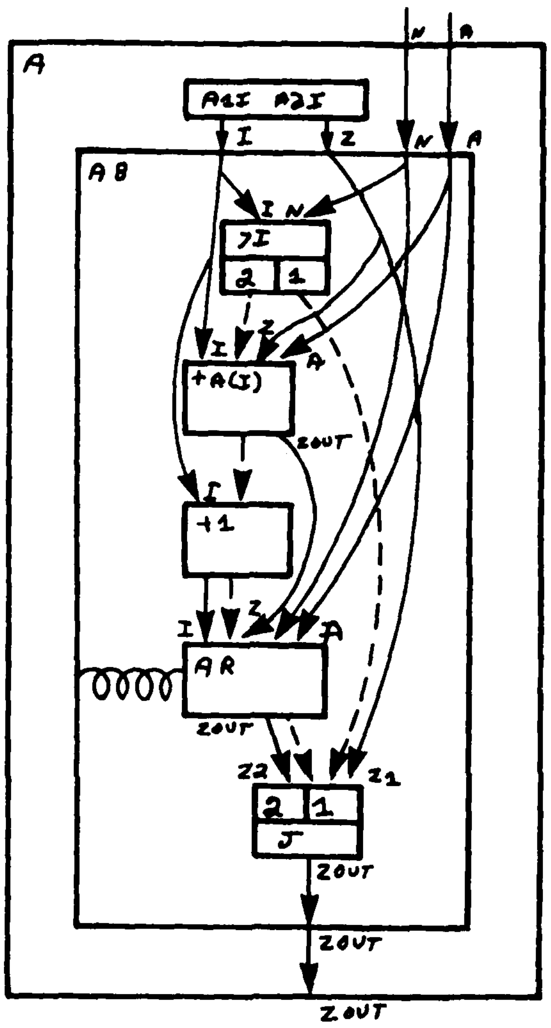
IV.2.1.5.2 Determining a Behavioral Description for a Temporal Composition

The great advantage of the PBM temporal composition is that it is easy to develop a behavioral description for the result. If 'emporal data flow is considered to be basically the same as ordinary data flow, then the subsegments of a temporal composition are connected together like a compositional group. (The possibility of multiple exits will be discussed below.) Since it is a compositional group, all of the methods discussed in Section IV.1.1 can be brought to bear on the problem of developing a behavioral description for the result. A proof of correctness for a temporal composition follows exactly the same lines as a proof for any compositional group.

```
                   Z = 0;
                   DO I=1 TO N;
                      Z = Z+A(I);
                   END;

          Segment A (the whole program above)
           inputs: A, N
   pre-conditions: REAL(A), VECTOR(A), INTEGER(N), 0≤N≤SIZE(A)
          outputs: ZOUT
  post-conditions: REAL(ZOUT), ZOUT=∑_{j=1,N} A(j), terminates(A)
        data flow: A.N→T.N, A.A→A2.A, A2.ZOUT→A.ZOUT
     control flow: T1→A
   temporal dflow: A1.TIOUT→T.TI, T.TJ→A2.TI
   justifications: (A2.po2)→A.po1, (A1.po2, A1.po3, T1.c1, T1.po1)→A.po3,
                   (A2.po3, A1.po2, A1.po3,1T.po2, T1.c1, T1.po3)→A.po2
              pbm: temporal composition
            roles: augmentation1 A1, termination1 T, augmentation2 A2

          Segment A1
 temporal outputs: TIOUT=(A1B.I at A1Bi)
  post-conditions: ∀i∈TIOUT(INTEGER(TIOUT_i)), ∀i∈TIOUT(TIOUT_i=i), ¬terminates(A1)

          Segment T
           inputs: N
  temporal inputs: TI=(>1.I at >1i)
   pre-conditions: INTEGER(N), ∀i∈TI(INTEGER(TI_i))
 temporal outputs: TJ=(TI_i at >I2o)
  post-conditions: ∀i∈TJ(TJ_i=TI_i), ∀i∈TJ(TJ_i≤N),
   justifications: (A.pr3)→T.pr1, (A1.po1)→T.pr2
       CASE1
       conditions: ∃i∈TI(TI_i>N)
  post-conditions: terminates(T), last-state(TI)=MIN({i∈TI| TI_i>N}),
                   last-state(TJ)=last-state(TI)-1
       CASE2
       conditions: ¬∃i∈TI(TI_i>N)
  post-conditions: ¬terminates(T), last-state(TJ)=last-state(TI)
```



Temporal Flow Diagram for Figure IV-23



Flow Diagram for Figure IV-23

```
          Segment A2
           inputs: A
  temporal inputs: TI=(+A(I).I at +A(I)i)
   pre-conditions: REAL(A), VECTOR(A), ∀i∈TI(INTEGER(TI_i)), ∀i∈TI(TI_i≤SIZE(A)),
                   last-state(TZ)=last-state(TI)+1,
                   TZOUT_last-state(TZOUT)=TZ_last-state(TZ)
          outputs: ZOUT
 temporal outputs: TZ=(A2B.Z at A2Bi), TZOUT=(ZOUT at A2Bo)
  post-conditions: ¬terminates(A2), REAL(ZOUT),
                   ZOUT=Σ_j=1,last-state(TZ)-1 A(TI_j)
   justifications: (A.pr1)→A2.pr1, (A.pr2)→A2.pr2, (A1.po1, T.po1)→A2.pr3,
                   (A.pr4, T.po2)→A2.pr4, (T)→A2.pr5, (T)→A2.pr6
```

Figure IV-23: An example of the PBM temporal composition.

As an example of all this, Figure IV-23 duplicates Figure III-9. The flow diagram in the figure shows the single self recursion which is equivalent to the temporal composition illustrated in the temporal flow diagram. In order to see how the behavioral description was derived, first consider what happens to the pre-conditions of the subsegments. As usual, the pre-conditions which are not internally satisfied become pre-conditions of the outer segment. The segment A1 has no pre-conditions. The pre-condition "INTEGER(N)" of T moves up. The pre-condition "$\forall i \in TI(INTEGER(TI_i))$" is satisfied by the first post-condition of A1. The first two pre-conditions of A2 move up. The third pre-condition of A2 is again satisfied by the first post-condition of A1 because T simply passes through the values it sees. The fourth pre-condition of A2 "$\forall i \in TI(TI_i \leq SIZE(A))$" is moved up to the outer segment. When this is done, the post-condition "$\forall i \in TJ(TJ_i \leq N)$" of T is used in order to convert the pre-condition into the form "$N \leq SIZE(A)$" which does not mention internal values. The last two pre-conditions of A2 are satisfied by the fact that there is a termination in the temporal composition which will satisfy them when the result is built up.

The first post-condition "REAL(ZOUT)" of A follows directly from the third post-condition of A2. The third post-condition of A "terminates(A)" addresses the question of the total correctness of A. It can be derived from the post-conditions of case 1 of T. Post-condition 2 of A1, and the fact that A1 does not, of itself, terminate implies that case 1 of T is applicable because A1 will eventually generate the value N+1. The second post-condition of case 1 of T "last-state(TI) = $MIN(\{i \in TI| TI_i > N\})$" makes it possible to conclude that "last-state(T.TI)=N+1".

There are two things to notice about this deduction. First, it is not trivial. The claim here is that the deduction is being done by recognizing a special case. Just as it is possible to solve simple recurrence relations associated with augmentations by recognition, it is possible to recognize how the termination test in the example interacts with the sequence of integers generated by A1. The basic fact is that a sequence of integers counting up from one will eventually pass any fixed integer. As a result, the segment A is guaranteed to terminate, and does not have to have a case corresponding to the possibility of non-termination. Further, it can be concluded that the first number in the sequence which is greater than N is $TI_{N+1}=N+1$. The second thing to notice is that if N is negative, then the first number in the sequence greater than N is actually $TI_1=1$. As a result, last-state(TI) actually equals MAX(1,N+1). In order to simplify the following discussion, it is assumed that the programmer intended that last-state(TI)=N+1, or in other words that $0 \leq N$. This becomes a pre-condition of A.

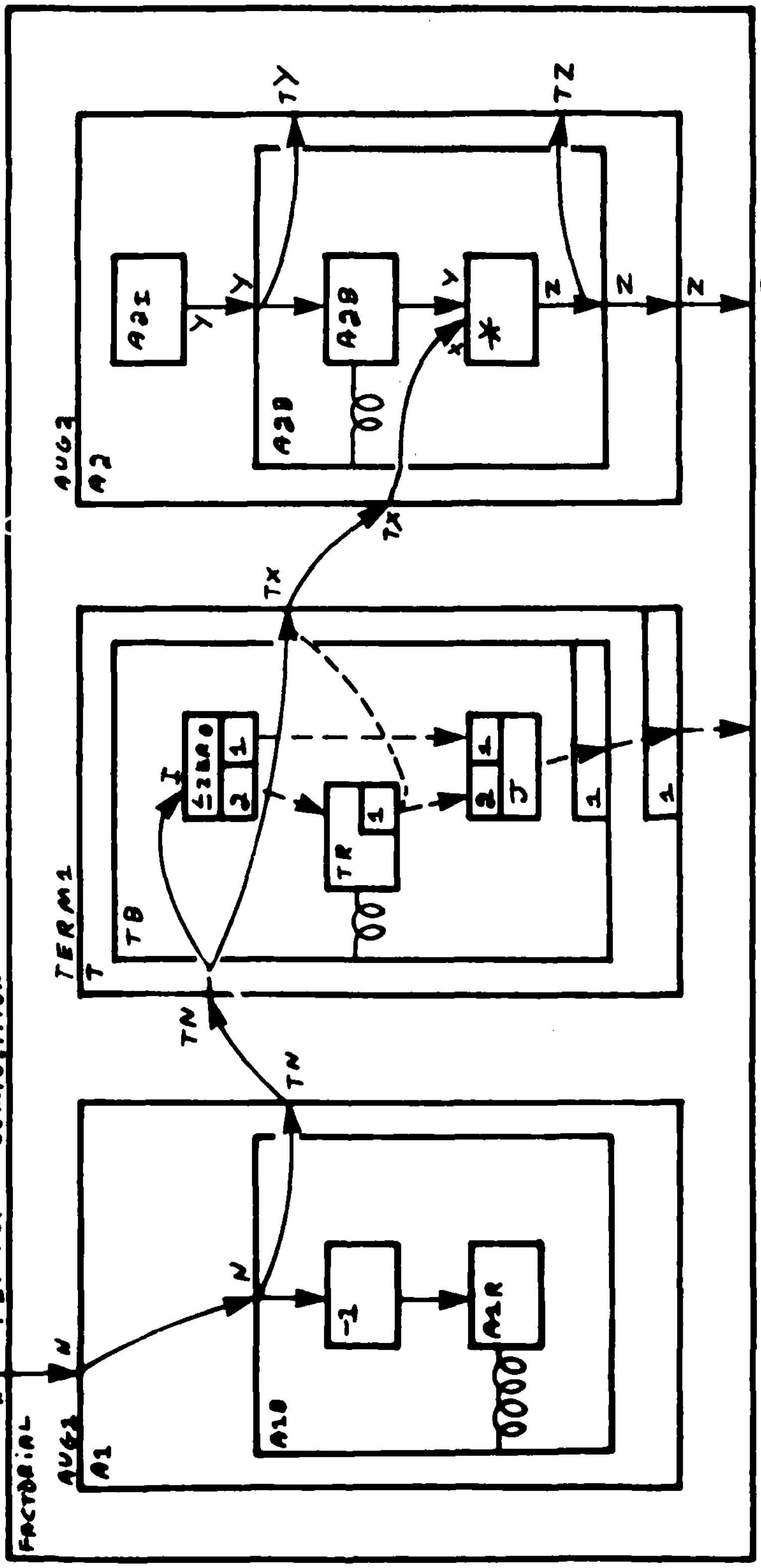
Once the last state of T.TI is known, the fact that temporal data flow constrains the ports it connects to have the same number of states, in conjunction with T1.po3, and A2.pr5, propagates this information to all of the temporal ports in the plan. It can be concluded that: last-state(A1.TIOUT) is N+1, last-state(T.TI) is N+1, last-state(T.TJ) is N, last-state(A2.TI) is N, last-state(A2.TZ) is N+1, and last-state(A2.TZOUT) is N+1. This propagation reflects the fact that the termination affects all of the other subsegments in the plan. It can be concluded that the outer segment A terminates because the way the PBM temporal composition combines its subsegments together ensures that if any of them terminates, then the result will terminate.

The second post-condition of A "$ZOUT=\sum_{j=1,N} A(j)$" can be derived by composing together the post-conditions of the subsegments. A2 asserts that "$ZOUT=\sum_{j=1,\text{last-state}(TZ)-1} A(TI_j)$". The second post-condition of A1 specifies that "$A1.TOUT_j=j$". The temporal data flow carries these values to A2 so that "$TI_j=j$". As discussed above, it can be concluded that "last-state(TZ)=N+1". Substitution leads to the post-condition for A.

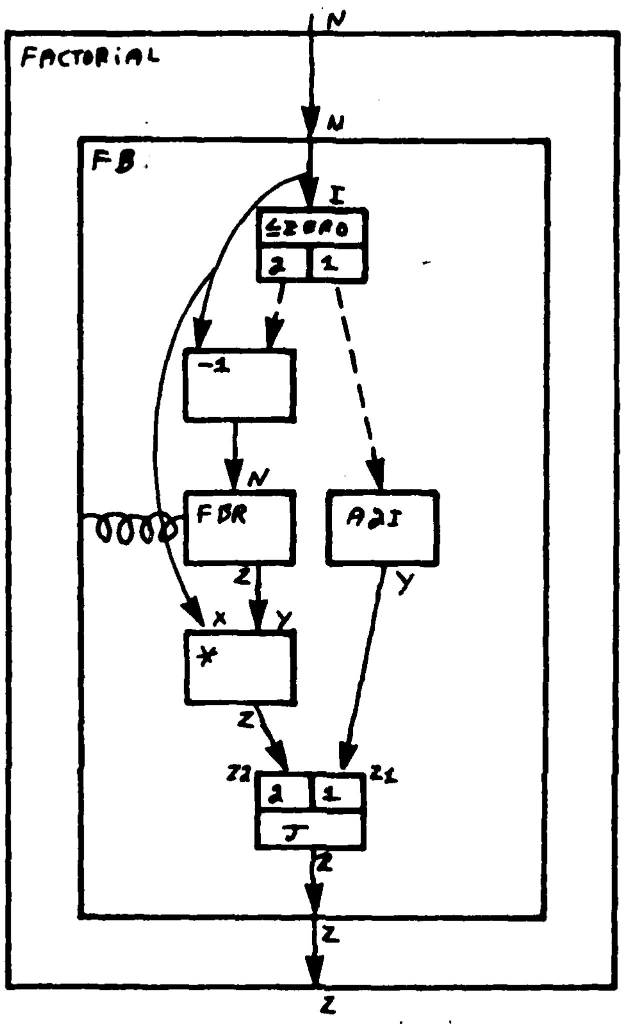
The program in Figure II-7 is very similar to the one above, with the addition of a filter. The filter takes in the truncated sequence of integers coming out of the termination, and further restricts it by selecting out only those integers which correspond to positive elements of A. The reduced sequence is then passed to the summation augmentation which adds up the corresponding elements of A. A behavioral description for that program could be derived in essentially the same way as in this example.

```
PROCEDURE FACTORIAL(N)
   IF N≤0 THEN RETURN(1);
         ELSE RETURN(N*FACTORIAL(N-1));
END;
```

```
           Segment A (the whole program above)
            inputs: N
    pre-conditions: INTEGER(N), N≥0
           outputs: Z
   post-conditions: INTEGER(Z), Z=$PROD_{j=1,N}$ j, terminates(A)
         data flow: A.N→A1.N, A2.Z→A.Z
      control flow: T1→A
    temporal dflow: A1.TN→T.TN, T.TNOUT→A2.TN
    justifications: (A2.po2)→A.po1, (A1.po1, A1.po2, T1.c1, T1.po1)→A.po3,
                    (A2.po3, A1.po2, A1.po3, T.po1, T1.c1, T1.po3)→A.po2
               pbm: temporal composition
             roles: augmentation1 A1, termination1 T, augmentation2 A2

           Segment A1
            inputs: N
    pre-conditions: INTEGER(N)
  temporal outputs: TN=(A1B.N at A1Bi)
   post-conditions: ∀i∈TN(INTEGER($TN_i$)), ∀i∈TN($TN_i$=1+N-i), ¬terminates(A1)
    justifications: (A.pr1)→A1.pr1

           Segment T
   temporal inputs: TN=(≤ZERO.I at ≤ZEROi)
    pre-conditions: ∀i∈TN(INTEGER($TN_i$))
  temporal outputs: TX=($TN_i$ at TR1o)
   post-conditions: ∀i∈TX($TX_i$=$TN_i$), ∀i∈TX($TX_i$>0)
    justifications: (A1.po1)→T.pr1
      CASE1
        conditions: ∃i∈TN($TN_i$≤0)
   post-conditions: terminates(T), last-state(TN)=MIN({i∈TN| $TN_i$≤0}),
                    last-state(TX)=last-state(TN)-1
      CASE2
        conditions: ¬∃i∈TN($TN_i$≤0)
   post-conditions: ¬terminates(T), last-state(TX)=last-state(TN)

           Segment A2
   temporal inputs: TX=(*.X at *i)
    pre-conditions: ∀i∈TX(INTEGER($TX_i$)), last-state(TY)=last-state(TX)+1,
                    $^{TZ}$last-state(TZ)=$^{TY}$last-state(TY)
           outputs: Z
  temporal outputs: TY=(A2B.Y at A2Bi), TZ=(A2B.Z at A2Bo)
   post-conditions: ¬terminates(A2), INTEGER(Z),
                    Z=$PROD_{j=1,last-state(TX)}$ $TX_j$
    justifications: (A1.po1, T.po1)→A2.pr1, (T)→A2.pr2, (T)→A2.pr3
```



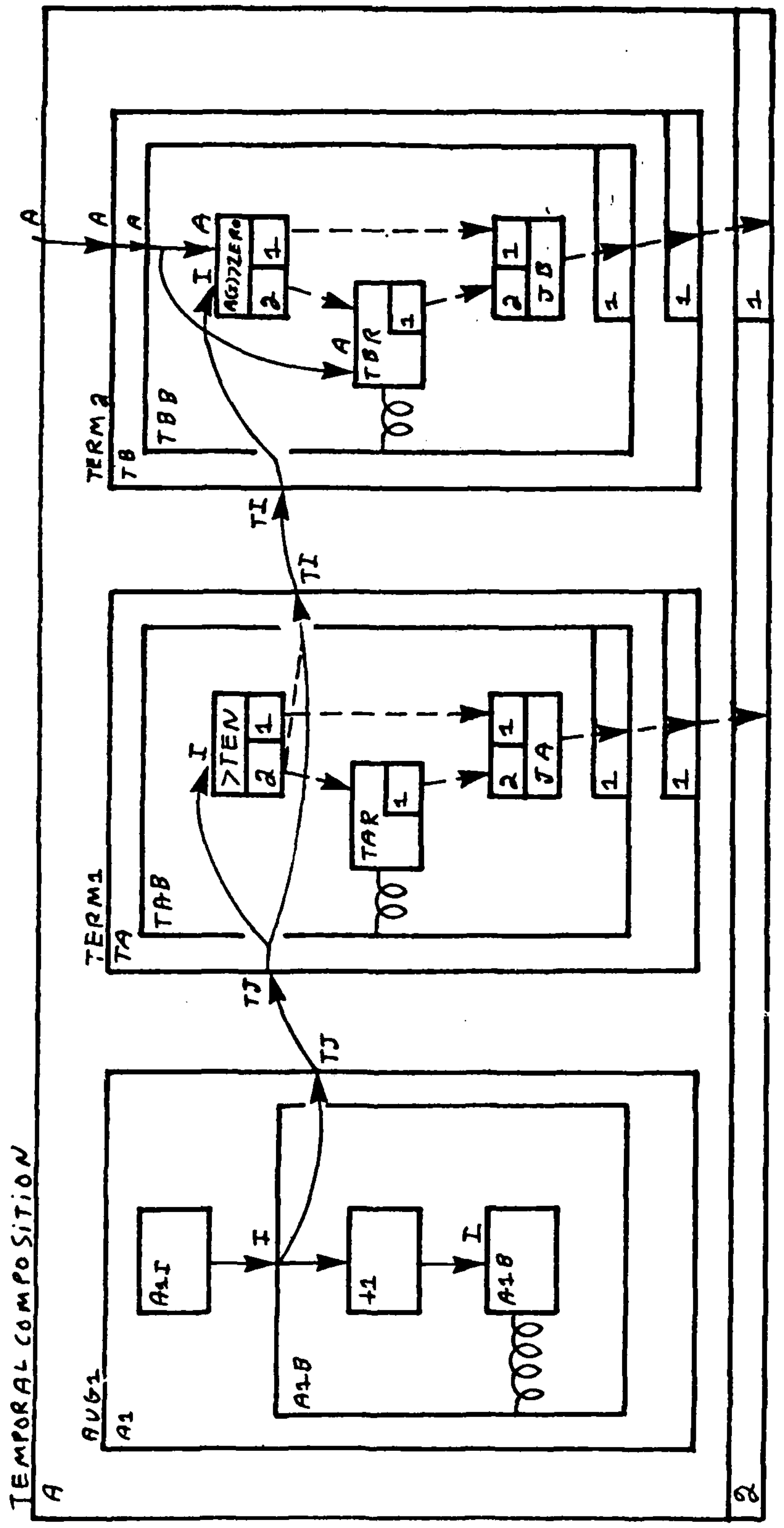
Temporal Flow Diagram for Figure IV-24



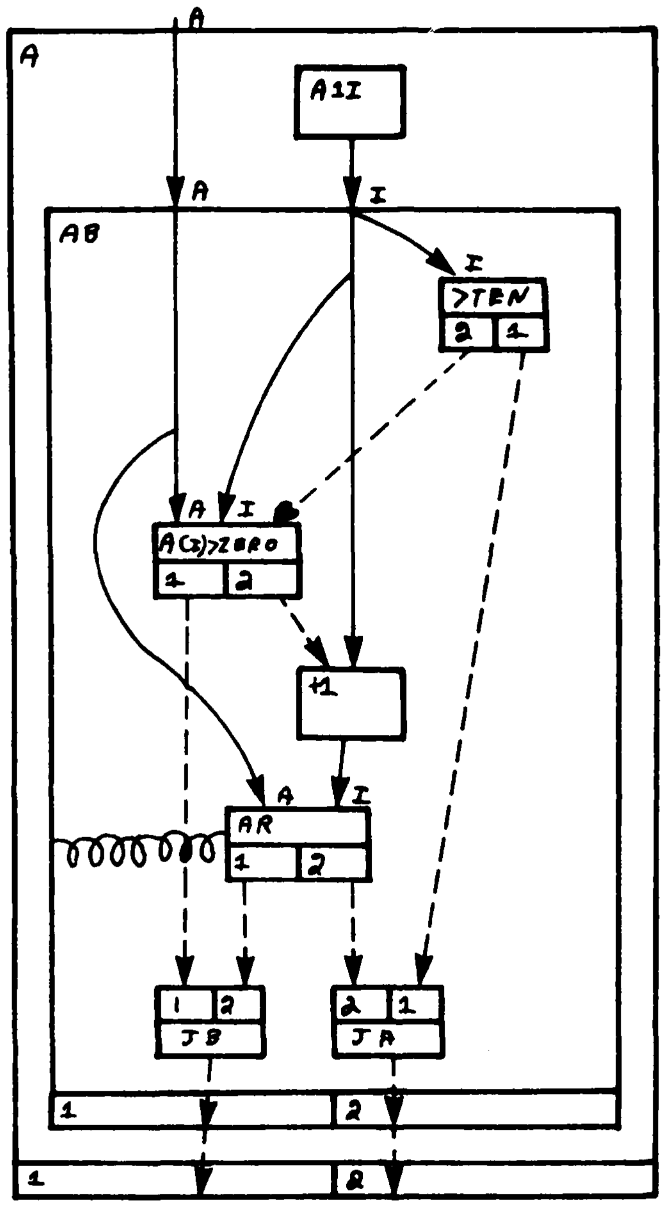
Flow Diagram for Figure IV-24

Figure IV-24: A temporal composition with a head recursive augmentation.

The temporal composition in Figure IV-24 has a head recursive augmentation A2. The augmentation A1 counts down from the input value N. It generates the sequence of values {N, N-1, N-2 , ...} where "TN_i=1+N-i". The termination truncates this sequence at zero. Recognition reveals that this occurs at TN_{N+1}=0 and that the result must terminate. As in the last example, this is supported by an assumption (here "N≥0") which becomes a pre-condition of the outer segment A. *The output TX of T passes through the values of TN which are seen at the output side of the* recurrence, namely {N,N-1, ... ,2,1}. Note that the temporal order in which they are seen at TR1o is the reverse of the temporal order in which they are seen at ≤ZEROi. The augmentation A2 computes the product of the sequences it receives, i.e. "Z=$PROD_{j=1,last\text{-}state(TX)}$ TX_j". Substitution yields "Z=$PROD_{j=1,N}$ 1+N-j" which can be rearranged as "Z=$PROD_{j=1,N}$ j" which is N factorial.



Temporal Flow Diagram for Figure IV-25



Flow Diagram for Figure IV-25

```
                    DO I=1 TO 10;
                       IF (A(I)>0) THEN GOTO FOUND;
                    END;
                    ...
             FOUND: ...

              Segment A (the whole program above)
               inputs: A
       pre-conditions: REAL(A), VECTOR(A), 10≤SIZE(A)
      post-conditions: terminal(A)
            data flow: A.A→TB.A
         control flow: TA1→A2, TB1→A1
       temporal dflow: A1.TJ→TA.TJ, TA.TI→TB.TI
       justifications: (A1.po2, A1.po3, TA1.c1, Ta1.po1)→A.po1,
                       (TB1.c1, Ta2.c1)→A1.c1, (TA1.c1, TB2.c1)→A2.c1
                  pbm: temporal composition
                roles: augmentation1 A1, termination1 TA, termination2 TB
         CASE1
           conditions: ∃i(1≤i≤10 ∧ A(i)>0)
         CASE2
           conditions: ∀i(1≤i≤10 → A(i)≤0)

              Segment A1
     temporal outputs: TJ=(A1B.I at A1Bi)
      post-conditions: ∀i∈TJ(INTEGER(TJi)), ∀i∈TJ(TJi=i), ¬terminates(A1)

              Segment TA
      temporal inputs: TJ=(>TEN.I at >TENi)
       pre-conditions: ∀i∈TJ(INTEGER(TJi))
     temporal outputs: TI=(TJi at >TEN2o)
      post-conditions: ∀i∈TI(TIi=TJi), ∀i∈TI(TIi≤10),
       justifications: (A1.po1)→TA.pr1
         CASE1
           conditions: ∃i∈TJ(TJi>10)
      post-conditions: terminates(T), last-state(TJ)=MIN({i∈TJ| TJi>10}),
                       last-state(TI)=last-state(TJ)-1
         CASE2
           conditions: ∀i∈TJ(TJi≤10)
      post-conditions: ¬terminates(T), last-state(TI)=last-state(TJ)

              Segment TB
               inputs: A
      temporal inputs: TI=(A(I)>ZERO.I at A(I)>ZEROi)
       pre-conditions: REAL(A), VECTOR(A), ∀i∈TI(INTEGER(TIi)), ∀i∈TI(TIi≤SIZE(A))
       justifications: (A.pr1)→TB.pr1, (A.pr2)→TB.pr2,
                       (A1.po1, TA.po1)→TB.pr3, (TA.po2, A.pr3)→TB.pr4
         CASE1
           conditions: ∃i∈TI(A(TIi)>0)
      post-conditions: terminates(T), last-state(TI)=MIN({i∈TI| A(TIi)>0}),
         CASE2
           conditions: ∀i∈TI(A(TIi)≤0)
      post-conditions: ¬terminates(T)
```

Figure IV-25: A temporal composition with two terminations.

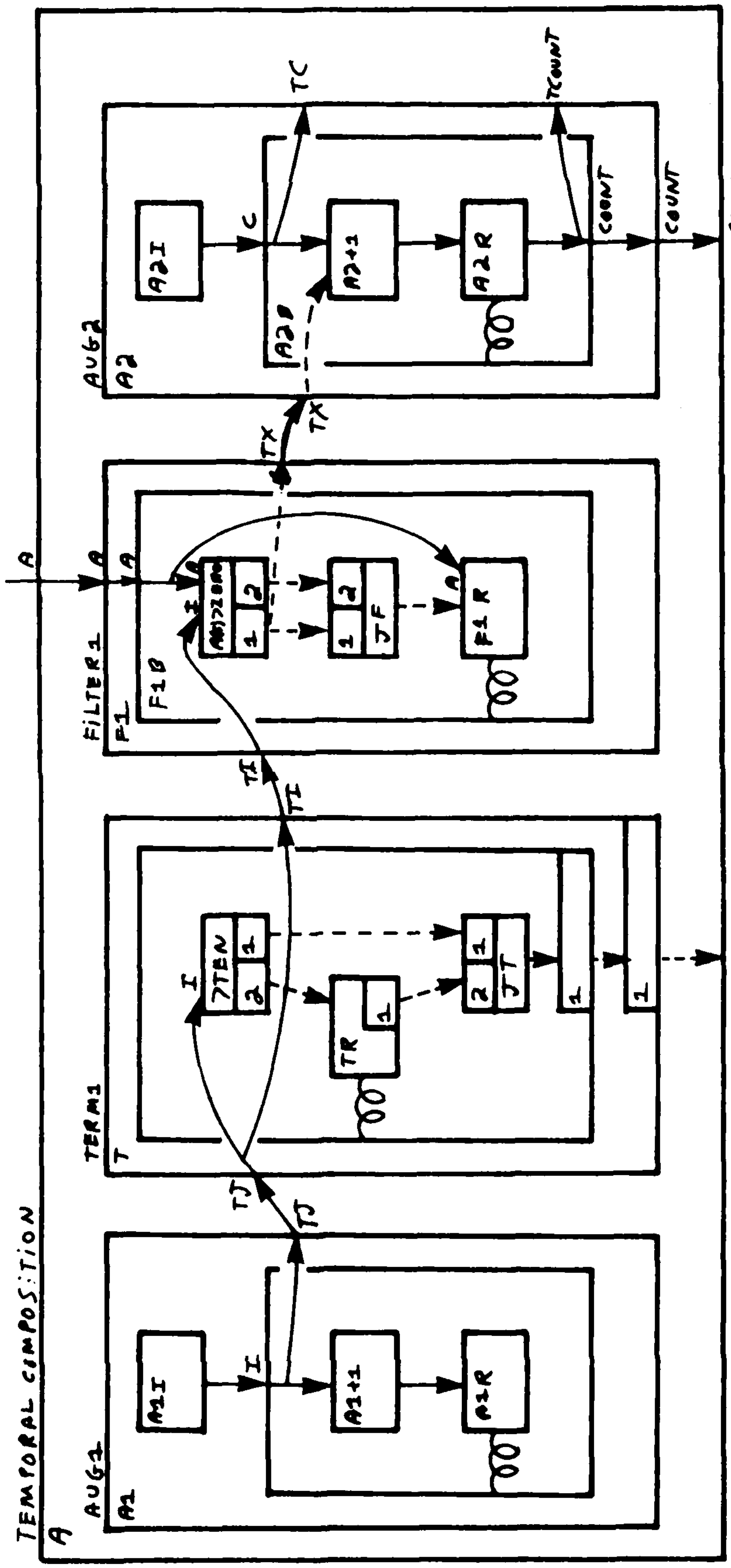
The temporal composition in Figure IV-25 has two terminations, and corresponds to the same program as in Figure IV-9. The augmentation A1 counts up from one. The interaction of A1 with the termination TA is clear. TA will cause A to terminate if TJ gets to $TJ_{11}=11$. If TJ does not reach TJ_{11}, then TA will not cause termination. This analysis is sufficient to show that A must terminate because it cannot continue on forever without reaching TJ_{11}.

In order to understand what the program does, it is looked at from two different points of view. It must either terminate due to TA or due to TB. The way the terminations interact in a temporal composition, it is impossible for them to both trigger termination at the same time. Here TA takes precedence over TB because TA prevents TB from seeing any values of I which cause TA to terminate.

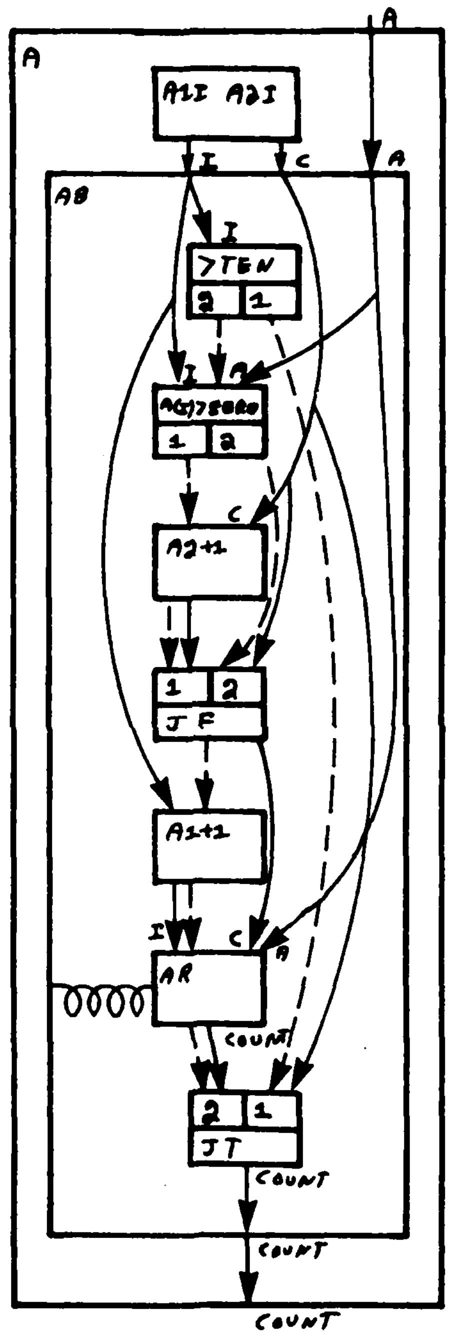
Assume A terminates due to TA. This means that "last-state(TJ)=11" and therefore "last-state(TI)=10". Further, it must be the case that TB does not terminate. Therefore, "$\forall i \in TI(A(TI_i) \le 0)$" or, substituting, "$\forall i(1 \le i \le 10 \rightarrow A(i) \le 0)$". Note that once it has been assumed that just one termination is causing termination, a multi-termination temporal composition, like an ordinary temporal composition, can be viewed as just a simple composition. If TB causes termination and TA does not, then "last-state(TJ)≤10" and "$\exists i \in TI(A(TI_i) > 0)$" or, substituting, "$\exists i(1 \le i \le 10 \wedge A(i) > 0)$".

```
COUNT=0;
DO I=1 TO 10;
   IF (A(I)>0) THEN COUNT=COUNT+1;
END;
```

```
          Segment A (the whole program above)
           inputs: A
   pre-conditions: REAL(A), VECTOR(A), 10≤SIZE(A)
          outputs: COUNT
  post-conditions: INTEGER(COUNT), COUNT=number-of(i| 1≤i≤10 ∧ A(I)>0),
                   terminates(A)
        data flow: A.A→F1.A, A2.COUNT→A.COUNT
     control flow: T1→A
   temporal dflow: A1.TJ→T.TJ, T.TI→F1.TI, F1.TX→A2.TX
   justifications: (A2.po2, F1.po1, A1.po2, A1.po3, T1.c1, T1.po3)→A.po2,
                   (a2.po1)→A.po1, (A1.po2, A1.po3, T1.c1, T1.po1)→A.po3
              pbm: temporal composition
            roles: augmentation1 A1, termination1 T, filter1 F1, augmentation2 A2

          Segment A1
 temporal outputs: TJ=(A1B.I at A1Bi)
  post-conditions: ∀i∈TJ(INTEGER(TJ_i)), ∀i∈TJ(TJ_i=i), ¬terminates(A1)

          Segment T
  temporal inputs: TJ=(>TEN.I at >TENi)
   pre-conditions: ∀i∈TJ(INTEGER(TJ_i))
 temporal outputs: TI=(TJ_i at >TEN2o)
  post-conditions: ∀i∈TI(TI_i=TJ_i), ∀i∈TI(TI_i≤10)
   justifications: (A1.po1)→T.pr1
        CASE1
       conditions: ∃i∈TJ(TJ_i>10)
  post-conditions: terminates(T), last-state(TJ)=MIN({i∈TJ| TJ_i>10}),
                   last-state(TI)=last-state(TJ)-1
        CASE2
       conditions: ∀i∈TJ(TJ_i≤10)
  post-conditions: ¬terminates(T), last-state(TI)=last-state(TJ)
```



Temporal Flow Diagram for Figure IV-26



Flow Diagram for Figure IV-26

```
          Segment F1
           inputs: A
  temporal inputs: TI=(A(I)>ZERO.I at A(I)>ZEROi)
   pre-conditions: REAL(A), VECTOR(A), ∀i∈TI(INTEGER(TI_i)), ∀i∈TI(TI_i≤SIZE(A))
 temporal outputs: TX=(at A(I)>ZEROlo)
  post-conditions: last-state(TX)=number-of(TI_i| TI_i>0), ¬terminates(A)
   justifications: (A.pr1)→F1.pr1, (A.pr2)→F1.pr2,
                   (A1.po1, T.po1)→F1.pr3, (A.pr3, T>po2)→F1.pr4

          Segment A2
  temporal inputs: TX=(at A2+1i)
   pre-conditions: last-state(TC)=last-state(TX)+1,
                   TCOUNT_last-state(TCOUNT)=TC_last-state(TC)
          outputs: COUNT
 temporal outputs: TC=(A2B.C at A2Bi), TCOUNT=(A2B.COUNT at A2Bo)
  post-conditions: INTEGER(COUNT), COUNT=last-state(TX)
   justifications: (T)→A2.pr1, (T)→A2.pr2
```

Figure IV-26: A temporal composition with degenerate temporal data flow.

The purpose of Figure IV-26 is to show an example of degenerate temporal data flow. The augmentation A1 counts up from one. The termination T truncates this sequence at ten. The filter F1 selects out those integers which correspond to positive elements of the vector A. The augmentation A2 counts up from zero. The temporal data flow from F1.TX to A2.TX is a degenerate one. It does not specify any input values for A2. A2 does not require any inputs. Rather it signifies only that the augmentation function A2+1 in A2 is executed in the execution environment created by the filter. Thus, F1 controls how many times the augmentation function will be executed. Analysis of A2 shows that "COUNT=last-state(TX)". Substitution, reveals that "COUNT=number-of(i| $1 \le i \le 10 \wedge A(I)>0$)".

The main point of these examples is that they are simple. The PBM temporal composition reduces the problem of understanding a single self recursive program almost entirely to the problem of understanding a set of augmentations, filters, and terminations. The sections discussing these PBMs showed that they, in turn, are usually, though not always, straightforward to understand. The next section shows that it is straightforward to analyze a program in terms of the PBM temporal composition.

Before looking at analysis, another issue needs to be addressed. That is the question of whether or not the above proofs based on temporal abstraction are applicable to the program which is produced by removing the temporal abstraction. The problem is that the temporal data flow and the subsegments referred to in the proof do not appear in the same form in the result.

The steps in a proof based on temporal abstraction can refer to data flow, control flow, temporal data flow, and parts of the behavioral descriptions of the subsegments. The steps will be applicable in the result, as long as, all of the claims made by these things are still accurate when the subsegments are combined. There is no problem with the non-temporal data flow and control flow because they do appear in the result in the same form.

The way a temporal data flow is handled when subsegments are combined ensures the continued validity of its two claims: that the execution environments of the two ports will be the same, and that the data items will be transmitted. The destination segment is placed in the same execution environment as the source. Ordinary data flow is added which transmits the required values. As more things are added in, neither of these relations is upset. It is not possible tc affect data flow unless it is rerouted, and the only rerouting which is ever done is through joins which preserve the original path while satisfying one of the pre-conditions of the subsegment in question. The only way the execution environments of segments are affected as the result is built up is that a new segment can be put between two old segments. If the new segment is a straight-line segment, then there is no problem. It becomes part of the old execution environment. When a filter is added in, the split and join together act like a straight-line segment. The only problem which could arise is involved with termination tests. If a termination test is put between two segments, then they will no longer be in the same execution environment. This may make no difference, but it would be a problem if a temporal data flow had linked the two segments. This problem is avoided by prohibiting the placement of a termination from contradicting any temporal data flow. This is essentially a constraint on the relationship between the temporal data flow entering the termination and the rest of the temporal data flow in the plan.

The last issue concerns the behavioral descriptions of the subsegments. Note first that all of the internal structure (control flow, data flow, and subsegments) of the subsegments does appear in the result, even though the subsegments themselves do not. The only changes which are made are those which are done in order to satisfy pre-conditions of the subsegments. The only data flow which is added between the subsegments is that which implements the temporal data flow.

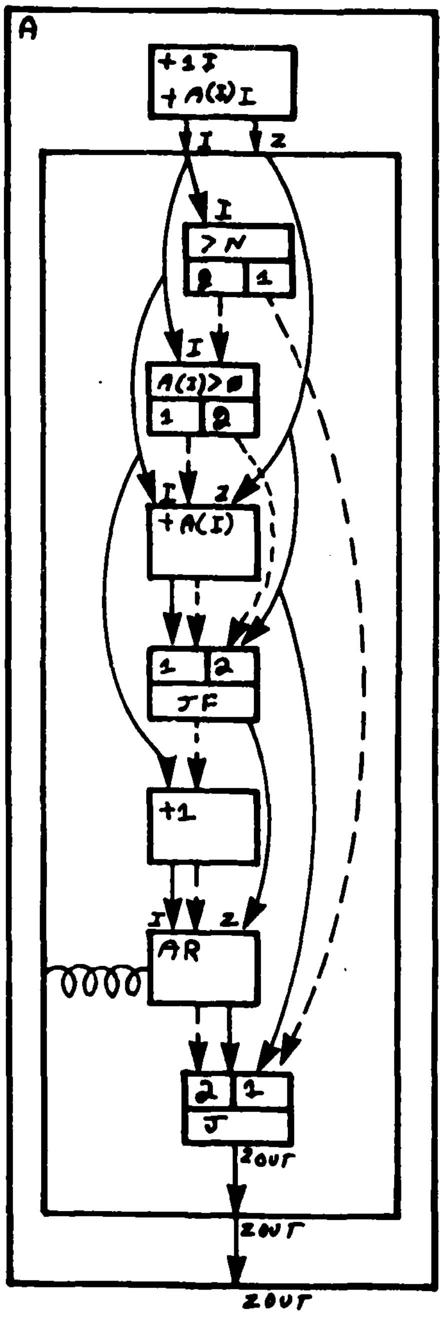
This can be looked at inductively. The combination process begins with a nucleus which has a behavioral description. When an augmentation or filter is added into the nucleus, this has absolutely no affect on anything that was in the behavioral description of the original nucleus. This is true because there is no data flow from the additional stuff to the original stuff, and there has been no effective change in the control flow between the parts of the old stuff. As a result, from the point of view of the original stuff, nothing has happened. The only data flow from the old stuff to the new stuff is that which implements the temporal data flow. The only change in the data flow in the new stuff is that which is done to satisfy pre-conditions of the new stuff. The only change in the control flow in the new stuff is that the augmentation function or filter test will be placed either before or after any termination test in the old stuff. All of these changes are things which are explicitly expected by the subsegment being added in, and are taken into account in the behavioral description. As a result, from the point of view of the new stuff, the only things which have happened are things that it expected to happen.

Adding in a termination is more complex. There is, as above, no problem with data flow, or from the point of view of the new stuff. However, the control flow in the original stuff is changed. Some of these changes merely reflect pre-conditions about the relative number of states in temporal ports, and are expected. The only other change is that the result may be capable of terminating sooner. This affects the absolute number of states in the temporal ports. However, all of the behavioral descriptions are valid no matter how many states there are because they are parameterized by the number of states in their temporal ports, and make only relative claims about the last states of ports. A behavioral description of a subsegment in a temporal composition is not allowed to make any absolute statements about the number of states in a temporal port precisely because the addition of a termination could invalidate any such claim.

It can be seen that there are a variety of restrictions on the subsegments and their interconnection which are needed in order to ensure that a proof based on subsegments will always carry over to the result. From the point of view of analysis, things are not so difficult. All that is needed is that the temporal composition which is produced by analyzing a program be such that the proof will carry over. This can be verified by seeing that all of the features of the original program are accurately represented in the temporal composition.

IV.2.1.5.3 Analyzing a Program in Terms of the PBM Temporal Composition

Once a piece of a program has been analyzed as a single self recursion, it can be further analyzed as a temporal composition. The analysis procedure is motivated by a desire to break the program up into pieces which are loosely coupled so that they can be understood in isolation from each other. The basic notion behind the analysis method is that a segment (A) can affect another segment (B) only if either A has data flow to B, or if A is a split, and the control flow out of A controls whether or not B is executed.



Flow Diagram for Figure IV-27

```
Z = 0;
DO I=1 TO N;
   IF A(I)>0 THEN
      Z = Z+A(I);
END;
```

Figure IV-27: An example single self recursive program.

Consider the flow diagram in Figure IV-27. Note that in the plan, the references to A and N have been treated like literals (segments without inputs which return the desired values) rather than like inputs to the program in order to simplify the data flow in the figure. The segment +A(I) has data flow going to itself through the port Z of AB. It has data flow going through join JF to itself, and data flow going to the output side of A through join J. However, it does not affect any other segment in the single self recursion A. (Note that when analyzing a single self recursion, the affects of a segment on a join can usually be ignored, however, see the discussion of Figure IV-33) The segment +A(I) can therefore be removed from segment A without changing anything that the other segments in A are doing. This observation forms the basis for analyzing a single self recursion as a temporal composition. The original single self recursion is decomposed by removing groups of segments which do not affect any other segments in the single self recursion. This is applied repeatedly to break the program up into a set of pieces. Each time a group of segments is removed, it is made into an augmentation or a filter. The analysis process is the exact reverse of the combining process described in Section IV.2.1.5.1.

The plan in Figure IV-28 shows the result after removing segment +A(I). The result R is a temporal composition of what remains in A and an augmentation (A1) constructed with +A(I) as its augmentation function. Note that the affects of +A(I) on the joins JF and J have been eliminated by removing the relevant data flow. The associated initialization is found by locating those parts of the initialization for A which have data flow only to the new augmentation being created. The temporal data flow shows what values +A(I) sees at its input I. The pre-condition "last-state(TZ) = last-state(TI)+1" reflects the fact that +A(I) was situated after the termination test (segment >N) in A. The pre-condition "$TZOUT_{last\text{-}state(TZOUT)} = TZ_{last\text{-}state(TZ)}$" reflects the fact that data flow through the join J in A caused the last result of +A(I) to become an output of segment A. These three devices (the temporal data flow, and the two pre-conditions) are needed because though +A(I) does not affect the rest of segment A, the rest of A does affect +A(I). Fortunately the effects on +A(I) can be summarized by means of these devices so that the actions of +A(I) in A can be understood in isolation from the rest of A.

Looking at what remains in A, it can be seen that the segments A(I)>0 and JF as a group do not affect any other segment in A. They used to affect +A(I) by controlling when +A(I) got executed, but it is now gone from A. Figure IV-29 shows the result after removing this filter. The temporal output connected to A1 (which has the same structure shown in Figure IV-28) is taken out as part of the filter (F1) because it refers to an execution environment controlled by the filter. Note that if it were not for the need to refer to this execution environment, the filter test and join could be grouped together into a single segment and treated as an augmentation function. The need to refer to an internal execution environment is what distinguishes a filter from an augmentation.

Looking at the residue remaining after removing the filter, reveals that each segment affects the other. There is data flow from +1 to >N and >N controls when +1 is executed. At this point, a second principle is used to control the decomposition. This is based on the idea that though >N

$TZOUT_{last\text{-}state(TZOUT)} = TZ_{last\text{-}state(TZ)}$
$last\text{-}state(TZ) = last\text{-}state(TI)+1$

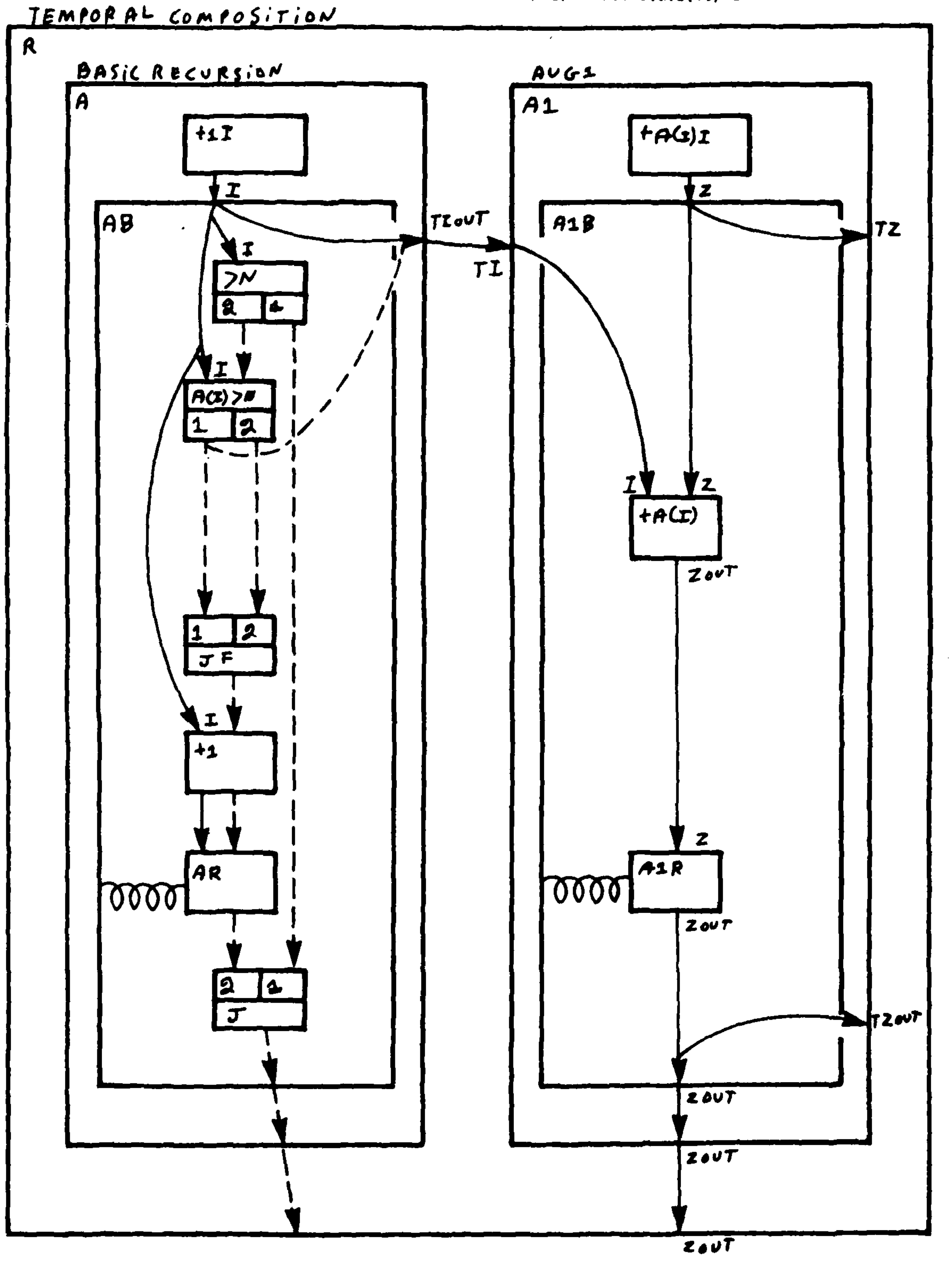


Figure IV-28: An example of removing an augmentation.

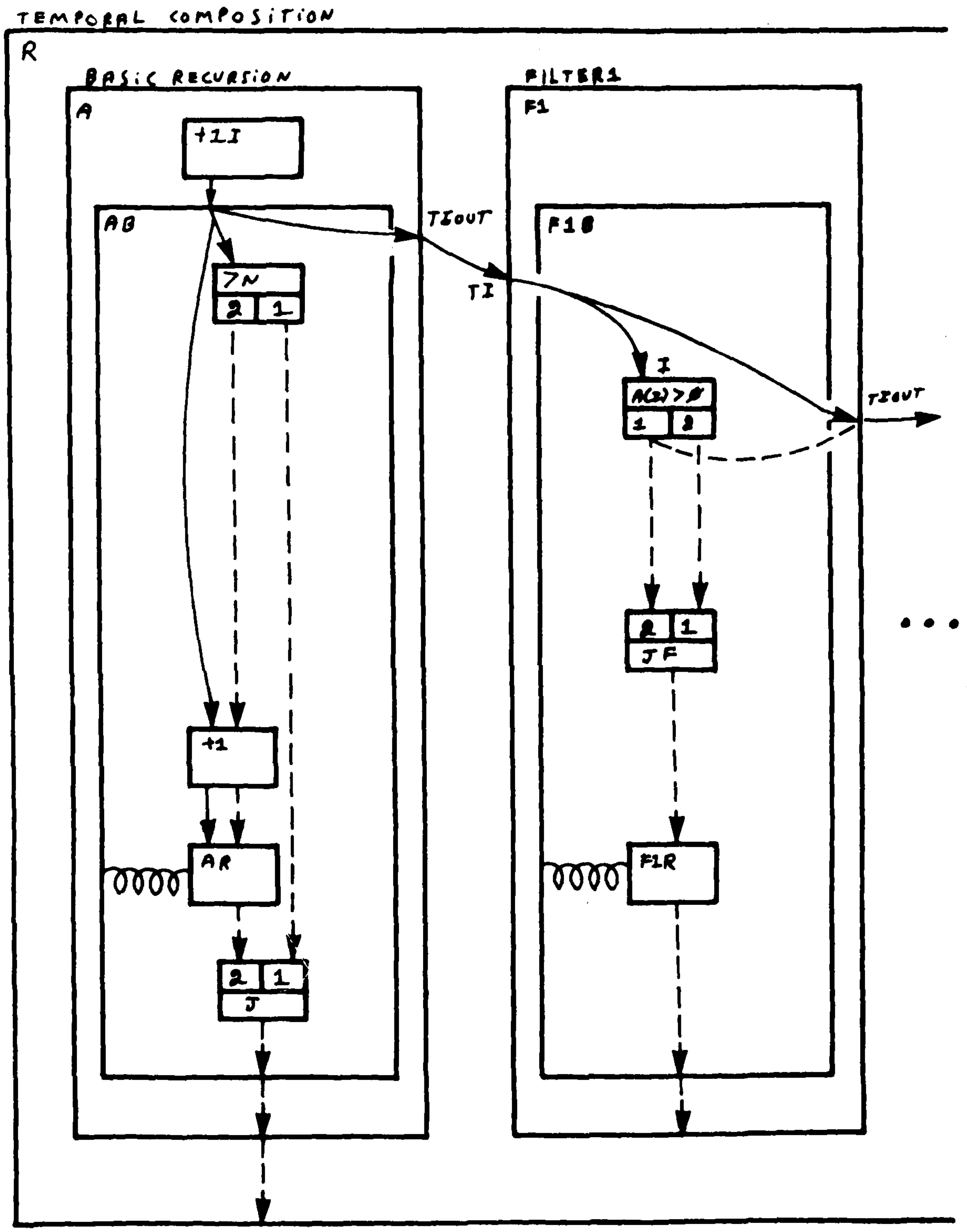


Figure IV-29: An example of removing a filter.

affects +1, it does so only in a very restricted way. It merely controls how many times +1 will be executed. It does this in a way which is much simpler than a filter test because all it can do is stop executing +1 after some number of times. It cannot skip an execution and then continue on later the way a filter can. The second principle says that a split and its associated join can be removed from a single self recursion and made into a termination as long as they do not have data flow to any other segment, and as long as all but one of the cases of the split cause termination of the single self recursion.

The parts removed become a termination as shown in Figure IV-30. Unfortunately, since the termination test did affect the rest of A, the rest of A is changed when it is removed. This is reflected by the fact that looking at a subsegment of a temporal composition in isolation, it is not possible to come to any conclusion about the absolute number of states associated with its temporal ports. The pre-conditions and post-conditions of the subsegment are prevented from making any absolute statements about how many states there are so that there will not be any erroneous conclusions drawn when it is looked at in isolation.

The residue left in Figure IV-30 after removing the termination is itself an augmentation and does not need to be analyzed further. Note that a pre-condition for this augmentation reflects where the termination test was situated with respect to it. Figure IV-31 shows the temporal composition which results from the analysis above.

Looking back at Figure IV-27, it can be seen that the segments A(I)>0, +A(I), and JF as a group do not affect any of the other segments in A. These three segments could be grouped together and taken out as an augmentation. This is an example of the fact that there are usually many different ways in which a single self recursion can be broken up as a temporal composition. Another example is the fact that the segment +A(I) can be broken into two separate augmentations: A conjunction augmentation A(I) which converts the sequence of integers into a sequence of elements of the vector A, and an augmentation +_ which adds up these elements. The heuristic which is used in order to decide what pieces to break a single self recursion up into is a compromise between breaking it up into as many pieces as possible which makes the pieces simple, but can lead to a very large number of pieces, and breaking it up into a small number of pieces which can lead to very complicated pieces. The heuristic breaks up the single self recursion up as much as possible, except that conjunction augmentations (like A(I) above) are combined into the pieces which use their outputs if possible.

This heuristic leads to the example program being broken up into four pieces as shown in Figure IV-31. The program could be broken up into as many as eight pieces as shown in Figure IV-32. The more expanded analysis leads to a more expanded explanation of what the program does as follows. There is a conjunction augmentation (1) which generates a sequence of integers all of which are equal to one. The augmentation +_A adds these up producing the partial sums {1,2, ...}. The termination >N terminates the program truncating the sequence of partial sums to {1,2, ... ,N}. This goes into a conjunction augmentation A(I)1 which converts it into a sequence of elements of A {A(1),A(2), ... ,A(N)}. This goes into a filter >_ which selects the positive elements of the vector A, and passes through the integers corresponding to these elements. The selected integers then go to A(I)2 which forms the sequence of the corresponding elements of A. (Note that this second use of A(I) is here because it is used twice in the program. A more efficient program would have only done one array reference.) Finally the augmentation +_B sums up the positive elements of A returning the total as its result. The heuristic above pulls the four conjunction augmentations into the segments they have temporal data flow to, reducing the number of last-state(TIOUT) = last-state(TI)+1

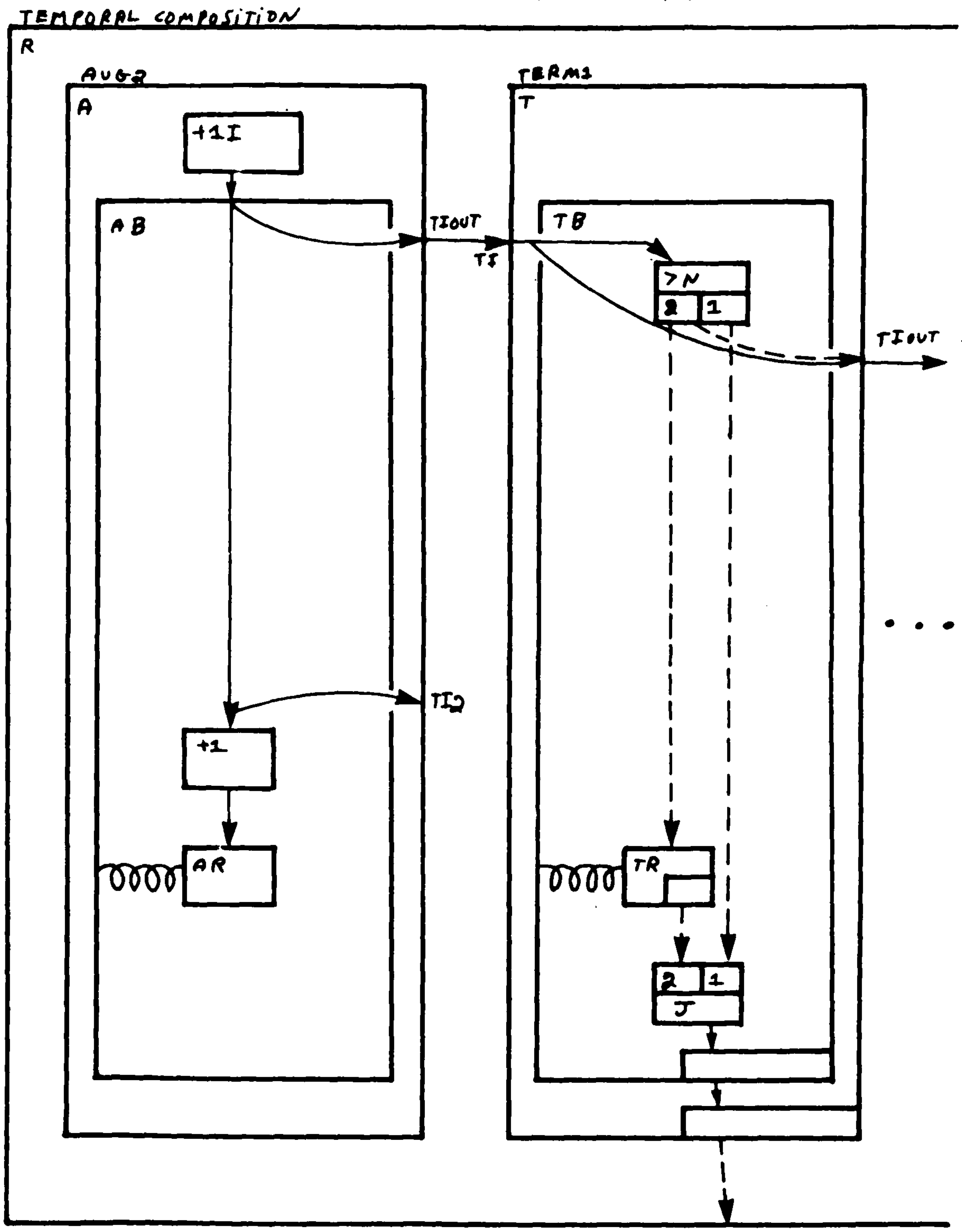


Figure IV-30: An example of removing a termination.

subsegments in the plan to four.

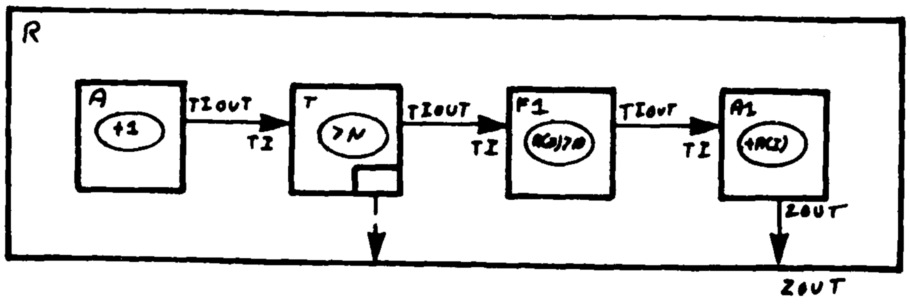


Figure IV-31: The complete temporal composition.

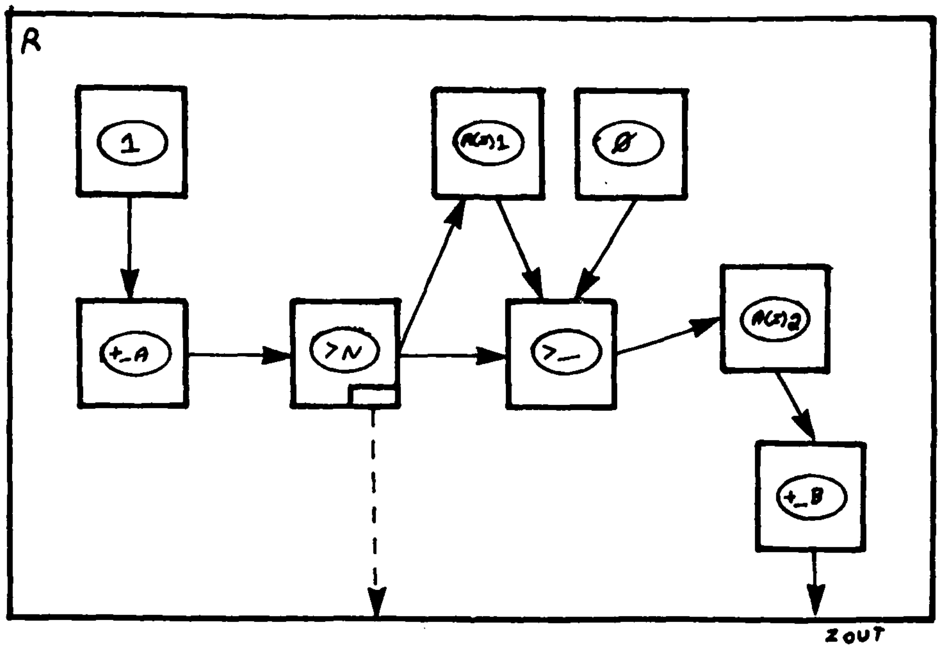


Figure IV-32: An alternate temporal composition analysis.

As further examples of analyzing a single self recursion, consider the programs in the last section. The program in Figure IV-23 is just the same as the example above, except that it does not have the filter. The plan in Figure IV-24 shows a program with some computation after the recurrence. In order to analyze it, the first thing that is noticed is that the segment * affects only itself. It is taken out as a head recursive augmentation. The segment A2I is recognized as its initialization because it has data flow only to * , and it is executed only once. After this, the termination ≤ZERO is removed leaving the augmentation -1.

Figure IV-25 shows a program with two terminations. In the plan, each segment affects the others. Therefore, no augmentations or filters can be removed. However, there are two terminations I>TEN and A(I)>ZERO which can be removed. The termination A(I)>ZERO is removed first because, on a given iteration of the program, I>TEN affects A(I)>ZERO by determining whether or not it will be executed, while A(I)>ZERO does not affect I>TEN. A(I)>ZERO does control whether I>TEN will be executed on the next iteration by controlling whether or not there will be a next iteration. Once A(I)>ZERO and join K are removed, then I>TEN and join J can be removed, which leaves +1 which is a simple augmentation. In general, if there is more than one termination, then the innermost termination is removed first.

The first thing to be removed when analyzing the plan in Figure IV-26 is the augmentation A2+1. When it is removed, a degenerate temporal data flow is introduced in order to show that the execution of A2+1 is controlled by the filter test A(I)>ZERO. If there had been any data flow from other parts of the program to A2+1, the temporal data flow created to represent it would have encoded this information. Once A2+1 is removed, the filter A(I)>ZERO, and then the termination >TEN are removed.

The purpose of figures IV-33 and IV-34 is to clarify a point about analyzing filters. It is not true that any split in the body of a single self recursion which cannot cause termination can be analyzed as a filter test. In order to take out the split X<0 in the figure as a filter, all of the computations it controls (namely -X) have to be taken out first. -X has data flow to +, so -X cannot be removed until after + is. There is no problem in removing +, as shown in Figure IV-34.

However, after + is removed, -X still cannot be removed. This is because the temporal port TX which receives values produced by -X is linked to the output port of join J. In the discussion above, when a segment to be removed was linked to a temporal port, the temporal port was simply removed with the segment (see Figure IV-29 for an example of this). Here, this is not possible. Note that the values of X seen at Jo are the absolute values of the elements in the vector A, while the values of X seen at -Xo are only the absolute values of the negative elements of the vector A. Data flow connecting to a port of a join can be ignored, only if there is not a temporal port referring to the port of the join.

-X cannot be removed. As a result, X<0 cannot be removed as a filter. However, the segments X<0, -X, and J as a group can be removed as an augmentation. The resulting augmentation is a conjunction augmentation which computes the absolute values of a sequence of numbers. The remaining program can then be analyzed by taking out A(I) and then >TEN.

In general, a split in a single self recursion can be analyzed as a filter test if and only if the following statements hold. The split must not be capable of terminating the single self recursion. If there is a segment whose execution is controlled by a case of the split, then the values it produces can only be used in two ways. They can become outputs of the single self recursion as a whole as in Figure IV-27. They can also be used by the segment itself, and other segments which are controlled by the same case of the split. The values cannot be used by a segment which is not controlled by the split, or which is controlled by a different case of the split.

To summarize, a segment of a program which has been analyzed in terms of the PBM single self recursion (see Section IV.2.1.1) can be further analyzed by the PBM temporal composition by repeatedly applying two principles. First, a group of segments can be removed from the single self recursion and made an augmentation or filter if they do not affect any other segment in the single self recursion. Second, a group of segments can be removed and made into a termination if the only affect they have on the rest of the single self recursion is to cause termination. If there are multiple terminations then the order in which they are removed is controlled by their order in the program. The process terminates when the residue cannot be further analyzed. If the residue itself is not an augmentation, a filter, or a termination, then the residue is put in the basic recursion role of the temporal composition. The heuristic about conjunction augmentations is used to reduce the number of pieces without significantly increasing their complexity. This process straightforwardly breaks up the single self recursion based solely on its data flow and control flow. The augmentations, filters, and terminations can then be understood separately, and their understandings

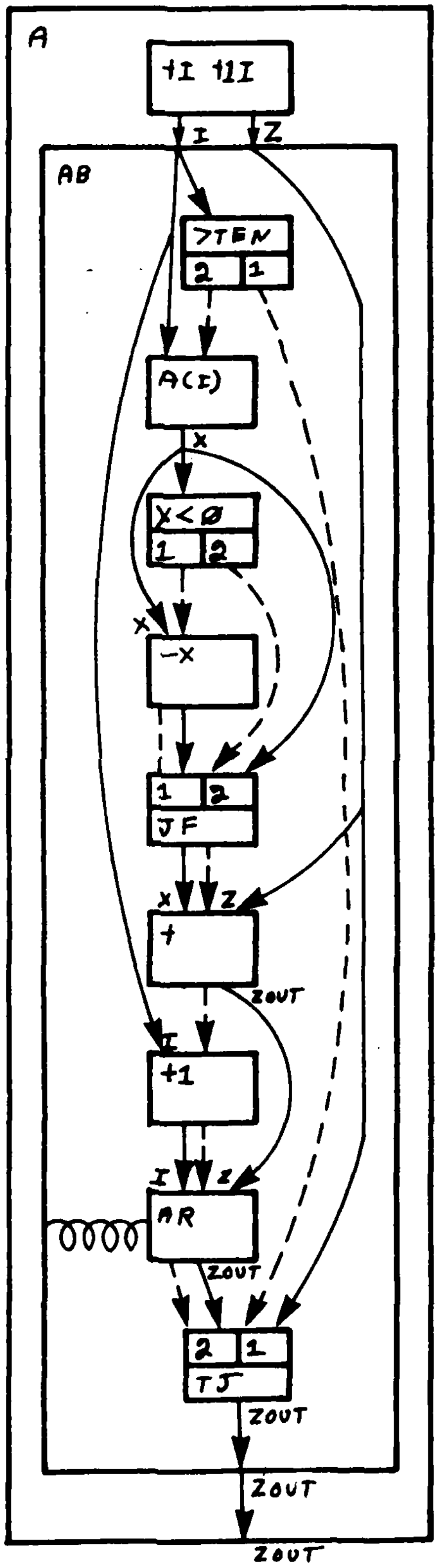


Figure IV-33: An example of a split which is not a filter.

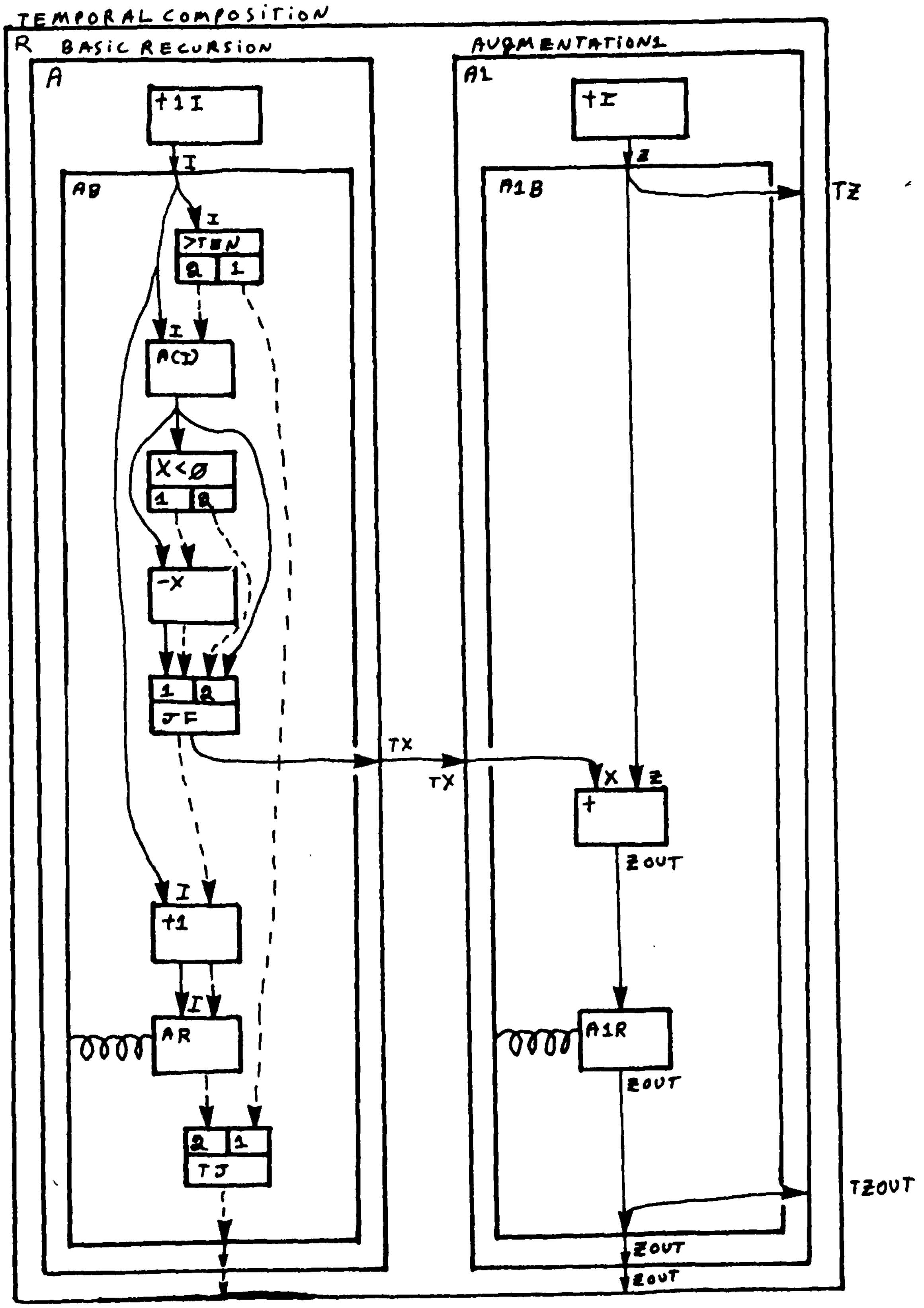


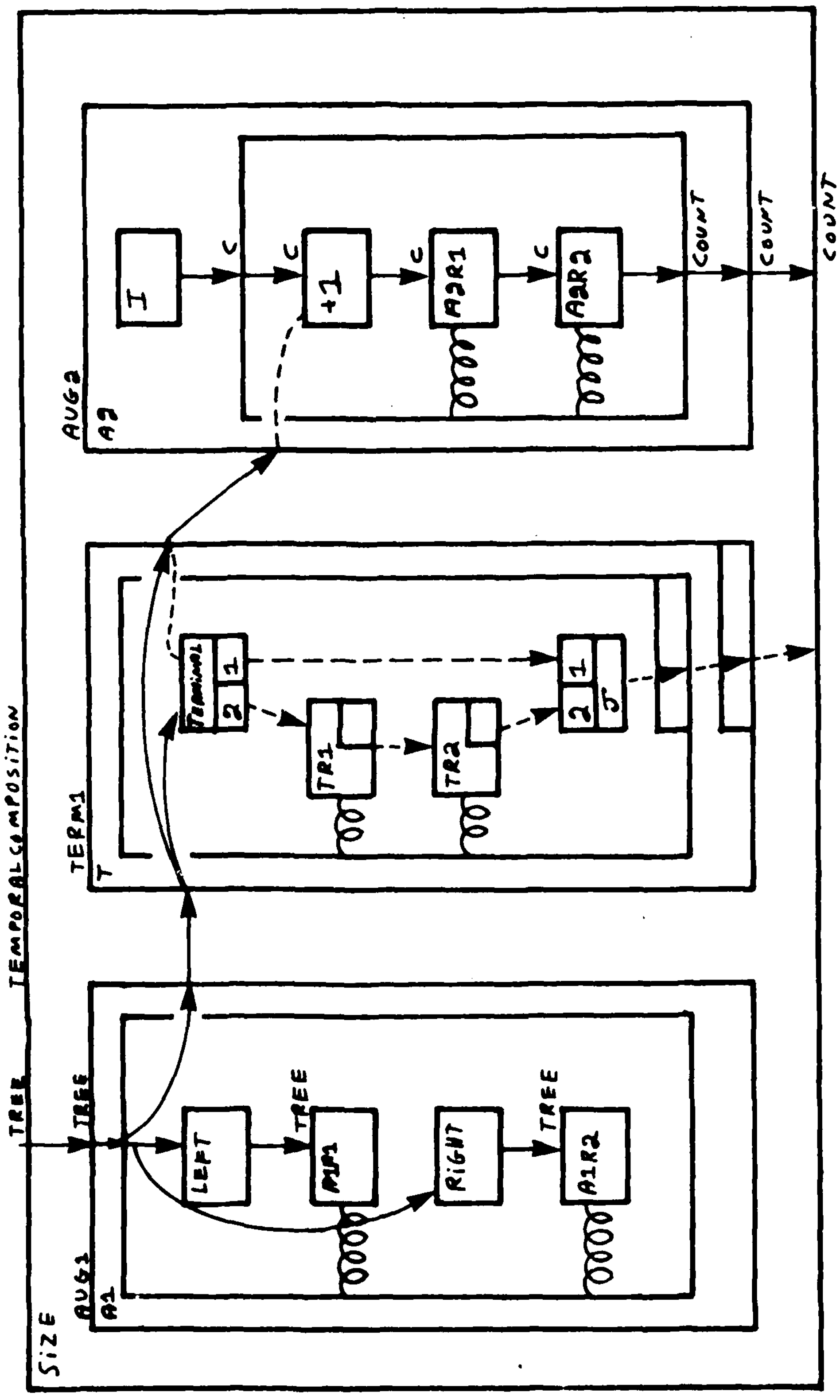
Figure IV-34: The situation after removing the segment +.

combined to produce an understanding of the whole single self recursion.

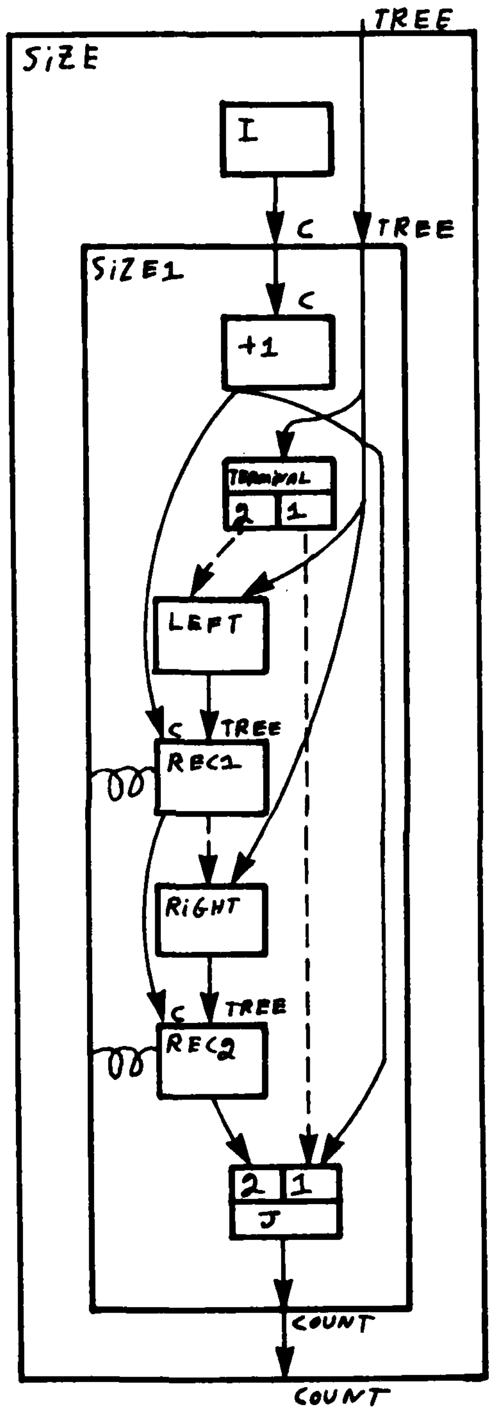
IV.2.2 Multiple Self Recursion

Single self recursive programs are restricted in that they can only call themselves once. If this restriction is relaxed, then the larger class of multiple self recursive programs can be considered. Many aspects of the PBMs discussed in the last section can be easily generalized to cover this larger class. However, several problems arise, and the resulting PBMs have not been worked out.

The PBM single self recursion can be directly extended by merely allowing there to be more than one recurrence. No other change is necessary. The PBM temporal composition can probably remain basically the same as far as the way the subsegments are interconnected, and the way the specifications are built up goes. However, what augmentations, terminations and filters look like has to be changed.



Temporal Flow Diagram for Figure IV-35



Flow Diagram for Figure IV-35

```
PROCEDURE SIZE(TREE);
   COUNT = 0;
   CALL SIZE1(TREE);
   RETURN(COUNT);
END;
PROCEDURE SIZE1(TREE);
   COUNT = COUNT+1;
   IF ¬TERMINAL(TREE) THEN DO; CALL SIZE1(LEFT(TREE));
                               CALL SIZE1(RIGHT(TREE)); END;
END;
```

Figure IV-35: An example of a multiple self recursive program.

The program in Figure IV-35 is an example of a multiple self recursive program. It computes the number of nodes, both terminal and non-terminal, in a binary tree. The analysis procedure described in Section IV.2.1.5.3 can be used to break the program up into three pieces. In order to be able to recombine the pieces as described in Section IV.2.1.5.1, all of the pieces need to have the same number of recurrences, in this case two. Up to this point, the entire PBM temporal composition has generalized in a nice way.

Consider the augmentation A2. It clearly has a lot in common with the single self recursive augmentation which counts up from zero. However, looked at closely, a couple of differences become apparent. First, there is the question of how to specify where the output COUNT comes from. The value of COUNT returned is the result of the last execution of +1. However, a pre-condition of the form "$TCOUNT_{last\text{-}state(TCOUNT)} = TC_{last\text{-}state(TC)}$" does not specify the right thing, and is in fact incorrect. Due to the data flow between the two recurrences, *A2R1.COUNT→A2R2.C, the relationship* between the values of TC and the values of TCOUNT is rather complex. Each time one of the branches of the recursion is terminated, the current value of TC becomes the value of TCOUNT and is propagated upwards. This brings up the second question which asks whether it is sufficient to associate states of temporal environments with integers in a double recursive situation like this. It might be better to associated each state with a two dimensional quantity such as a path in a tree, which corresponds to the structure of recursive calls. It would then be easier to specify the interaction between TC and TCOUNT. Unfortunately, changing the way the states are named obscures the commonality between A2 and a single self recursive counter. A different approach to handling A2 is to give it only one recurrence, in which case it would be an instance of a single self recursive counter. This makes it clear that A2 is in fact doing exactly the same thing as a single self recursive counter. However, the combining process would then have to be extended so that it could combine subsegments with different numbers of recurrences.

Unlike A2, the augmentation A1 is fundamentally different from a single self recursive augmentation. It has two augmentation functions, LEFT and RIGHT, in addition to two recurrences. The recurrence relations which describe what is going on in A1 are not compatible with states named by integers. Part of the problem is that as it stands, neither the augmentation function RIGHT, nor the recurrence A1R2 will ever by executed. If the states of TOUT *are named by* integers, none of the named states end up associated with A1R2. The augmentation A2 also has this problem. With A2 this problem can be eliminated by expressing the augmentation with one recurrence. A1 cannot be reasonably expressed using one recurrence, due to the interactions of the two augmentation functions with the two recurrences. Things would be even more complicated if one of the augmentation functions received data flow from one of the recurrences. The switch to more complex state names is forced. These problems aside, A1 does correspond to a useful segment of the program. It is a common augmentation which enumerates the nodes in a binary tree.

The termination T is different from a single self recursive termination. The key difference is that case 2 of the termination test, TERMINAL, is not executed just once. Rather it is executed many times in order to terminate each branch of the recursion. This points up the inappropriateness of the concept of last-state in this situation, and the need for a tree structured naming of states. A further complication which is possible in a multiply self recursive program is a test which bypasses some, but not all, of the recurrences. This construction does not correspond to either a termination, or a filter. The greatest problem with this construction is that the number of recurrences executed on a given iteration of the program becomes variable, rather than just all or none. That could make it very difficult to name the states in such a way that the names correspond to the structure of recursive calls.

In summary, it is likely that PBMs for multiple self recursive programs can be constructed which are extensions of the single self recursive PBMs. However, this has not been done yet. The most important thing which generalizes directly is the analysis procedure based on what segments affect other segments. It can break a multiply self recursive program up into loosely coupled parts which are meaningful in isolation. Unfortunately, these parts are more difficult to understand than single self recursive parts. The most basic change which is required is a switch away from naming states of temporal ports with integers. This allows the parts to be described, but adds to the complexity of the PBMs.

IV.2.3 Mutual Recursion

In the discussions above, recursive programs have been required to call themselves only directly. Relaxing this restriction allows mutually recursive programs. A fundamental difference this causes, is that once subroutines are allowed to be mutually recursive, they cannot be understood in isolation from each other. One way to understand them is to integrate them together into one large subroutine which is self recursive. However, this usually leads to a very complicated subroutine which does not reflect the logical structure of the program. Another approach is to generalize the single self recursive PBMs to cover mutually recursive programs, and use them to analyze the subroutines as a group. This should certainly be possible to some extent, however, the generalization would probably be more difficult and less successful than in the case of multiply recursive programs.

There is another way to look at recursive programs which is already embodied in the PBMs described above. In the description of the straight-line PBMs, nothing was said which prevents a subsegment from being a recursive instance of some other segment. This means that any program, even if it is singly, multiply, or mutually recursive, can be analyzed in terms of these PBMs. The knowledge associated with the recursive PBMs looks at a recursive program from the point of view of temporal sequences, and treats recurrences in special ways. This allows them to determine what a recurrence does, and reason about the termination of a program. Analyzing a recursive program with straight-line PBMs does not use the temporal sequence viewpoint, treats recurrences essentially the same as terminal segments (which implies that behavioral descriptions for them must be supplied from outside the system), and does not deal with the termination of the program. However, straight-line analysis can still be a very logical and useful way to look at a program.

To see the difference between these two ways of looking at a recursive program, consider the following program which computes the length of a list.

```
PROCEDURE LENGTH(LIST);
   IF EMPTY(LIST) THEN RETURN(0);
                  ELSE RETURN(1+LENGTH(TAIL(LIST)));
END;
```

If this program is analyzed in terms of the recursive PBMs, it is seen to be a temporal composition of an augmentation which enumerates the sublists of a list, a termination which stops this process when the empty list is encountered, and a head recursive augmentation which counts the number of sublists enumerated, and therefore computes the length. On the other hand, if the program is analyzed in terms of the straight-line PBMs, then it is seen to be a conditional. If its argument is an empty list, then it returns a length of zero. Otherwise, it returns one plus the length of the tail of the list. The program is clearly correct assuming that the recursive call does indeed compute the length of the tail of the list.

The recursive PBM analysis corresponds to a method of constructing recursive programs which is based on operating on temporal sequences. The straight-line analysis corresponds to a different method for constructing programs. One place where this method is often used is in the construction of a large set of programs. The method proceeds by first specifying what the programs are supposed to do, and then writing the individual programs. When each program is written, it is allowed to include calls on other programs from the set being written to perform subtasks. When a programmer is writing a program, how may not even be aware of whether or not the subroutine calls he is putting in the program are mutually recursive. This corresponds to the fact that in a straight-line analysis, the fact that some segment is a recurrence is basically incidental. A problem with this method of writing a set of programs is that the termination of the programs has to be shown separately, and this may not be easy. This aspect is also evident in the straight-line analysis itself.

The only way the current PBMs can deal with multiply and mutually recursive programs is through straight-line PBM analysis. There are some situations where this analysis reveals the logical structure the programmer had in mind. There are other situations where an analysis along the lines of the single self recursive PBMs would be more appropriate. It may well be possible to develop other useful classes of PBMs which can be applied to recursive programs in addition to these two.

IV.2.4 Analyzing Recursive Programs

Analyzing programs which contain loops and recursion is done in two phases. First individual instances of recursion are located, and then they are analyzed in terms of the PBMs temporal composition, augmentation, filter, and termination. This later process is discussed in detail in Section IV.2.1.5.3. As mentioned in the last two sections, more complex recursive programs, as such, are not dealt with by the current PBMs.

Two things should be noted about the process of analyzing a single self recursion. First, it is possible to construct a single self recursive program which cannot be analyzed in terms of the PBM temporal composition (except trivially) because every subsegment affects some other subsegment. (Any single self recursion can be trivially analyzed in terms of the PBM temporal composition by simply putting the entire single self recursion in the basic recursion role of the temporal composition.) Second there is no guarantee that the pieces which result from analyzing a single self recursion will be simple to understand in their own right. It turns out, however, that most of the time, things work well (see Chapter V).

In order to find the single self recursions in a program, the code for the program is analyzed in terms of the PBMs single self recursion, expression, conjunction, composition, predicate, and conditional. This is equivalent to a language where the flow of control is expressed solely by the nesting of operators in expressions, the sequential placement of statements, the possibly recursive calling of subroutines, the if-then-else construct, and an extended do-while construct which allows multiple exits from a loop. Analogous to the situation described in Section IV.1.4, any program can be expressed in such a language, and therefore any program can be analyzed in terms of the PBMs. However, the code for the program may have to be transformed when this is done. Whether or not transformations will have to be done depends on whether or not the pattern of control flow in the program is "reducible" as described by Allen and Cocke [1, 2]. They flow graphs they work with are essentially equivalent to a surface plan with all of the data flow arcs deleted, leaving only the control flow arcs.

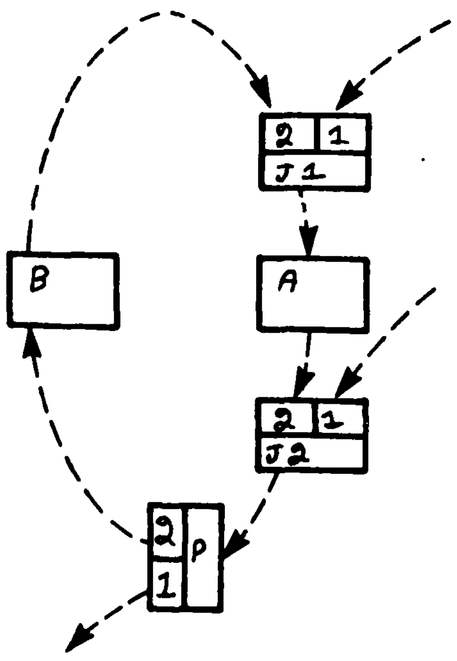


Figure IV-36: A multiple entry loop.

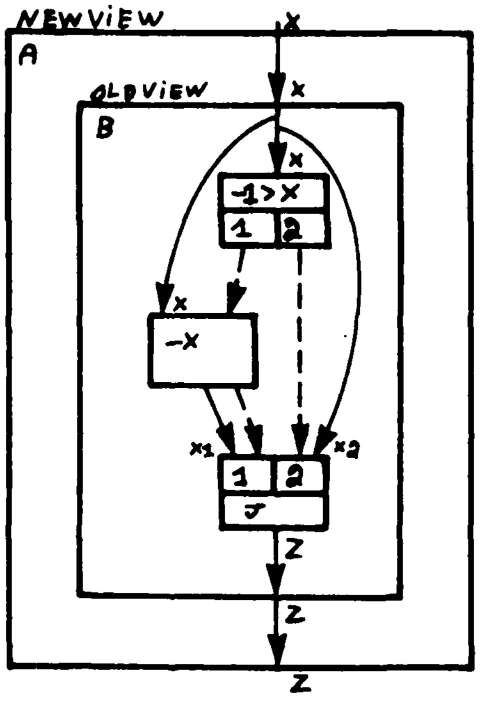
If the code for a program does not need to be transformed in order to be expressed in terms of if-then-else and extended do-while, then it can be easily analyzed in terms of PBMs by the system described in Section VI.2.1, whether or not it is actually expressed in terms of those constructs. The chief situation where transformations must be applied is a multiple entry loop such as the one shown in Figure IV-36. (Note that the control flow graph in the figure reduces to one of the basic irreducible flow graphs.) The PBM single self recursion cannot be applied to a loop with more than one entry point. Note that the key to the problem in the figure is the existence of the computation A. If it was not there, then the two joins could be combined, and the loop would have only one entry point. One way to transform the loop is to move segment A back across the join J1, and duplicate it so that it is executed both inside the loop before case 2 of J1, and outside the loop before case 1 of J1. The loop could then be recognized as:

```
ENTRY1: A
ENTRY2: DO WHILE P; B; A; END;
```

The two separate control flow paths leading to the loop would have to be recognized as part of a conditional. The current analyzer does not perform this type of transformation, and is not able to analyze a loop like the one in the diagram.

IV.3 The PBM New View

This PBM has just one role, named the old view, which must be a simple segment. There can be arbitrary acyclic subsegment data flow linking the ports of the result with the ports of the old view. As far as its structure in concerned, this PBM corresponds exactly to the trivial case of a conjunction with only one action. The purpose of the PBM new view is to associate a different behavioral description with a segment. It corresponds to a step in a proof about the program. Figure IV-37 shows an example of this PBM.



Flow Diagram for Figure IV-37

```
                    IF -1>X THEN Z=-X;
                           ELSE Z=X;

            global knowledge:
             ABS(Y)=(IF 0>Y THEN -Y IF 0≤Y THEN Y)

                  Segment A
                   inputs: X
           pre-conditions: INTEGER(X), EVEN(X)
                  outputs: Z
          post-conditions: INTEGER(Z), Z=ABS(X)
                data flow: A.X→B.X, B.Z→A.Z
           justifications: (B.po1)→A.po1, (A.pr2, B.po2, gk1)→A.po2
                      pbm: new view
                    roles: old view B

                  Segment B
                   inputs: X
           pre-conditions: INTEGER(X)
                  outputs: Z
          post-conditions: INTEGER(Z), Z=(IF -1>X THEN -X IF -1≤X THEN X)
           justifications: (A.pr1)→B.pr1
```

Figure IV-37: An example of the PBM new view.

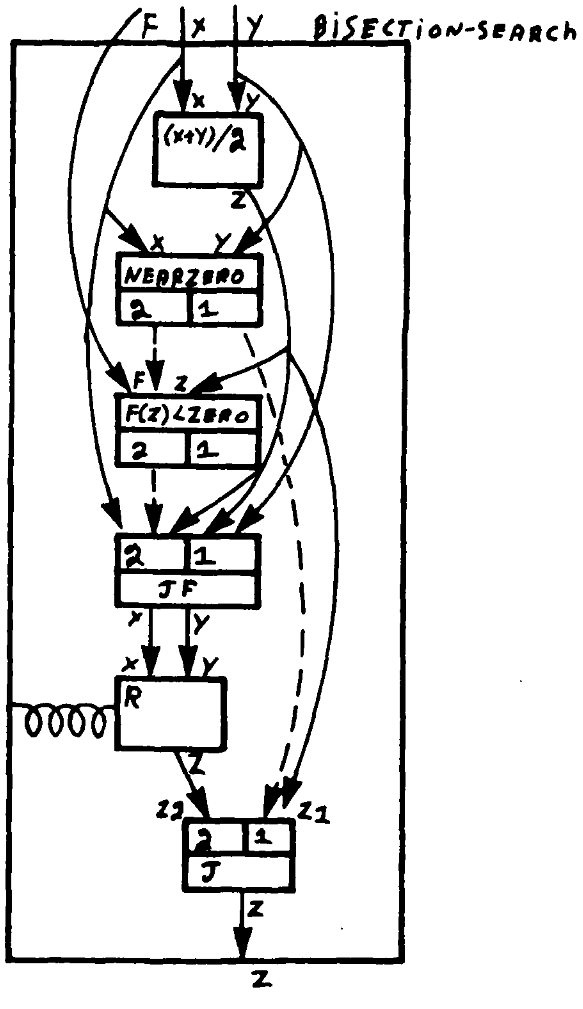
The old view in the figure, segment B, negates its input if it is less than -1. The behavioral description for B is typical of the kind of behavioral description generated for a conditional. The new view asserts that the output is the absolute value of the input. Note that this is only true due to the fact that A requires its input to be even, and therefore it cannot be -1. This example illustrates the two basic kinds of changes which the PBM new view can make in the behavioral description of a segment. First, it can simply state it in a different but equivalent form. Second, it can reduce the scope of the behavioral description by adding additional pre-conditions. Note that there is no way to determine that the behavioral description of A is appropriate by looking at B in isolation. Its appropriateness is a function of the larger context. A more interesting example of the PBM new view could be constructed by using the PBM to associate the improved behavioral description in Figure IV-5 with the composition in Figure IV-4.

The PBM new view is different from the other PBMs in that it is not found during the analysis of a program. Analysis proceeds based on the data flow and control flow in a program. From this point of view, the PBM new view is trivial and does not contribute to the breaking up of the program. Also, unlike the other PBMs, there is no procedure for determining the behavioral description of the result. The whole point is that the behavioral description of the new view is different from the behavioral description of the old view, and therefore different from what would be automatically generated. There is, however, a proof schema associated with the PBM. It must be shown that the pre-conditions of the old view follow from the pre-conditions of the new view, and that the post-conditions of the new view follow from the post-conditions of the old view, and the pre-conditions of the new view. In the example, this requires the piece of global knowledge indicated.

Rather than being introduced during analysis, instances of the PBM new view are introduced later on while working with the plan. Their primary use is to record a lemma about the program, namely that the new behavioral description is accurate. It should be noted that once the new view has been created, it can be used to help construct improved behavioral descriptions for the segments containing it. Instances of the PBM new view serve as bridges between the knowledge associated with the PBMs, and other knowledge about the program.

IV.4 The Meta-PBM instantiation

Instantiation is not itself a PBM, but rather a method for constructing PBMs. Any plan can be turned into a PBM by using it as a template. The terminal subsegments of the plan become roles of the PBM. The subsegments filling these roles are required to satisfy the behavioral descriptions of the corresponding terminal subsegments. The data flow and control flow is fixed. It is taken directly from the data flow and control flow in the plan.



Flow Diagram for Figure IV-38

```
        LOOP: Z=(X+Y)/2;
              IF Y-X<.0002 THEN GOTO DONE;
              IF F(Z)<0 THEN X=Z; ELSE Y=Z;
        DONE: ...

         Segment BINARY-SEARCH
          inputs: X, Y, F
  pre-conditions: REAL(X), REAL(Y), X<Y, FUNCTION(REAL→REAL,F),
                  NONDECREASING(F), CONTINUOUS(F), F(X)≤0≤F(Y)
         outputs: Z
 post-conditions: REAL(Z), ∃R(X≤R≤Y ∧ F(R)=0 ∧ ABS(Z-R)<.0001)
     subsegments: (X+Y)/2, NEARZERO, F(Z)<ZERO, JF, R, J
       data flow: BINARY-SEARCH.F→F(Z)<ZERO.F, BINARY-SEARCH.X→(X+Y)/2.X
                  BINARY-SEARCH.X→JF2.X2, BINARY-SEARCH.X→NEARZERO.X
                  BINARY-SEARCH.Y→NEARZERO.Y, BINARY-SEARCH.Y→(X+Y)/2.Y,
                  BINARY-SEARCH.Y→JF1.Y1, (X+Y)/2.Z→JF2.Y2,
                  (X+Y)/2.Z→JF1.X1, (X+Y)/2.Z→F(Z)<ZERO.Z,
                  (X+Y)/2.Z→J1.Z1, JF.X→R.X, JF.Y→R.Y,
                  R.Z→J2.Z2, J.Z→BINARY-SEARCH.Z
    control flow: NEARZERO2→F(Z)<ZERO1, NEARZERO1→J1,
                  F(Z)<ZERO2→JF2, F(Z)<ZERO1→JF1
  justifications: (R.po1, J2.po1, (X+Y)/2.po1, J1.po1)→BINARY-SEARCH.po1,
                  (R.po2, J2.po1, NEARZERO.c1, BINARY-SEARCH.pr6,
                   BINARY-SEARCH.pr7, (X+Y)/2.po2, J1.po1)→BINARY-SEARCH.po2

         Segment (X+Y)/2
          inputs: X, Y
  pre-conditions: REAL(X), REAL(Y)
         outputs: Z
 post-conditions: REAL(Z), Z=(X+Y)/2
  justifications: (BINARY-SEARCH.pr1)→(X+Y)/2.pr1,
                  (BINARY-SEARCH.pr2)→(X+Y)/2.pr2

         Segment NEARZERO
          inputs: X, Y
  pre-conditions: REAL(X), REAL(Y)
  justifications: (BINARY-SEARCH.pr1)→NEARZERO.pr1,
                  (BINARY-SEARCH.pr2)→NEARZERO.pr2
     CASE1
      conditions: Y-X<.0002
     CASE2
      conditions: Y-X≥.0002

         Segment F(Z)<ZERO
          inputs: Z, F
  pre-conditions: REAL(Z), FUNCTION(REAL→REAL,F)
  justifications: ((X+Y)/2.po1)→F(Z)<ZERO.pr1,
                  (BINARY-SEARCH.pr4)→F(Z)<ZERO.pr2
     CASE1
      conditions: F(Z)<0
     CASE
      conditions: F(Z)≥0

         Segment JF
         outputs: X, Y
     CASE1
          inputs: X1, Y1
 post-conditions: X=X1, Y=Y1
     CASE2
          inputs: X2, Y2
 post-conditions: X=X2, Y=Y2
```

```
          Segment R
           inputs: X, Y, F
   pre-conditions: REAL(X), REAL(Y), X<Y, FUNCTION(REAL→REAL,F),
                   NONDECREASING(F), CONTINUOUS(F), F(X)≤0≤F(Y)
          outputs: Z
  post-conditions: REAL(Z), ∃R(X≤R≤Y ∧ F(R)=0 ∧ ABS(Z-R)<.0001)
   justifications: (BINARY-SEARCH.pr1, JF2.po1, (X+Y)/2.po1, JF1.po1)→R.pr1,
                   (BINARY-SEARCH.pr2, JF1.po2, (X+Y)/2.po1, JF2.po2)→R.pr2,
                   (BINARY-SEARCH.pr3, (X+Y)/2.po2, JF1.po1, JF1.po2,
                    Jf2.po1, JF2.po2)→R.pr3, (BINARY-SEARCH.pr4)→R.pr4,
                   (BINARY-SEARCH.pr5)→R.pr5, (BINARY-SEARCH.pr6)→R.pr6,
                   (BINARY-SEARCH.pr7, F(Z)<ZERO1.c1, JF1.po1, JF1.po2,
                    F(Z)<ZERO2.c1, JF2.po1, JF2.po2)→R.pr7

          Segment J
          outputs: Z
        CASE1
           inputs: Z1
  post-conditions: Z=Z1
        CASE2
           inputs: Z2
  post-conditions: Z=Z2
```

Figure IV-38: The program binary-search.

The program, BINARY-SEARCH, in Figure IV-38 uses bisection of an interval in order to search for a root of a nondecreasing continuous function F. The program will terminate returning a value of Z between X and Y which is within .0001 of a root of F. The plan for this program can be converted into a PBM with six roles. For example, in the PBM BINARY-SEARCH, the (X+Y)/2 role is required to compute the average of X and Y. The F(Z)<ZERO role is required to determine whether or not the value of the function F at the point Z is negative. It does not matter how the subsegments perform their computations so long as they satisfy the behavioral descriptions. Once a subsegment is selected for each role, the resulting segment can be created by copying the structure of the plan. A behavioral description for the result can be generated by simply copying the behavioral description from the plan. Similarly, justification links for the result can be copied from the plan.

Note that the behavioral description for BINARY-SEARCH, and hence for the result is much more useful and informative than the behavioral description which would be generated by means of an analysis in terms of the straight-line and recursive PBMs. This is the most important feature of instantiation. It provides a way to connect up with more specific knowledge about an algorithm. BINARY-SEARCH could be analyzed as being a temporal composition of an augmentation consisting of (X+Y)/2, F(Z)<ZERO, and JF, and a termination consisting of NEARZERO, and J. Unfortunately, the recurrence relation describing what the augmentation does is not at all simple. As a result, the behavioral description constructed would be very unwieldy. A more fundamental problem is that there is nothing intrinsic to BINARY-SEARCH which would lead you to expect any more than that X and Y are real numbers, and that F is a function from reals to reals. As a result, these are the only pre-conditions which would be in the generated behavioral description. Given only these pre-conditions, BINARY-SEARCH is not guaranteed to find a root. In fact, there is no guarantee that F has any roots to find. This situation is similar to the situation with new views. The behavioral description associated with BINARY-SEARCH is more restrictive than the most general behavioral description for BINARY-SEARCH, and therefore there is no way to reasonably arrive at it by looking at BINARY-SEARCH in isolation.

Like the PBM new view, instantiation acts as a link between the general PBMs and more extensive knowledge about specific algorithms. The PA is designed to contain a library of knowledge about specific algorithms. What information should be in this library, and how it should be represented is being investigated by Charles Rich [80, 82].

When analyzing a program, it is preferable to recognize part of it as an instance of a particular algorithm about which a lot is known, rather than analyze it in terms of the more general PBMs. The problem with this is that such recognition is not easy. One of the major difficulties in a recognition task like this is that given a program, there are a lot of algorithms which might match parts of the program, and a lot of places in the program where each one might match. This can lead to an explosion of competing hypotheses. The analysis system described here (see Chapter VI) does not attempt to recognize instances of algorithms like BINARY-SEARCH. Rather, it analyzes a program solely in terms of the straight-line and recursive PBMs. It does not suffer from excessive hypotheses because there are only a small number of PBMs, and they are very different from each other. Charles Rich is investigating how to recognize instances of particular algorithms. In order to try and reduce the problem of competing hypotheses, he uses the output of the PBM analyzer as the input to his recognizer. The way the PBMs break up a program should reduce the number of hypotheses which have to be considered.

IV.5 Other Possible PBMs

Each PBM is intended to capture a method used in constructing programs. There are useful methods for constructing programs which are not covered by the PBMs above. One such method is the use of pure GOTOs. This is a rather unconstrained construct, and can lead to considerable complexity. However, it seems very reasonable in some situations. It would be difficult to develop a PBM which allows the introduction of arbitrary GOTOs. However, PBMs could be developed to cover the individual situations where pure GOTOs are reasonable.

One situation where GOTOs are reasonable is as error exits. In that situation, a GOTO is used to jump up several levels, and cause a premature exit. The following PBM might successfully embody this concept. The proposed PBM has two roles, a segment and a split. The split can be put arbitrarily far down inside the segment. A control flow from one of the cases of the split goes directly up to the output side of the segment, adding a new case to the segment. The definition of a plan would have to be relaxed in order to allow this direct control flow which hops over segment boundaries. It should be noted that the situation described by this new PBM can be described in terms of the straight-line and recursive PBMs, by making the control flow explicitly pass through a case of each intermediate segment. However, this may not do a good job of expressing the logical structure of the program, because the error exit really does not have anything to do with the intermediate segments. This is particularly clear when the GOTO crosses subroutine boundaries, and the segments the GOTO passes through can vary. As an example, consider a system which reads in requests from a user in the form of expressions which it evaluates in order to do what the user wants. Suppose that these expressions can refer to variables, and that there is a routine in the system which evaluates variables and causes an error exit to the top level if a variable is undefined. This can be modeled directly as an error exit. Alternatively, virtually every subroutine in the system would have to have an error case reflecting the fact that one of the subroutines it calls might call the variable evaluator. This alternative approach has poor modularity.

The behavioral description for the result of adding an error exit with the proposed PBM is derived as follows. The specifications for the original cases of the segment remain essentially unchanged. The only effect the error exit has on them is to introduce the possibility that these cases will not occur. The conditions of the old cases are extended by adding the requirement that the error exit condition not occur. The basic specifications for the new case are just conditions claiming that the error exit condition will occur. The pre-conditions of that case are taken from the pre-conditions of the old cases. Producing post-conditions about the partial state of the computation when th error occurs is difficult. Note that when analyzing a program this PBM makes it possible to separate out an error exit, and analyze the rest of the program as if it was not there. As an example of an error exit, consider a program which uses a hill climbing algorithm in order to find a solution to an equation. It might have an error exit which stopped the process if oscillation developed. The basic purpose of the error exit in this example is to guarantee that the program will terminate. The normal operation of the program can be understood by ignoring the error exit. When the error exit does occur, all that is known is that oscillation occurred. The other results are unreliable.

Another situation where GOTOs are very reasonable is in a program which is structured like a finite state automaton. One place this kind of control structure is used is in a parser. However, it can appear in any kind of a program which has a state. Usually, most of the state information in a program is stored in variables. However, it is sometimes useful to encode some of the state information into the flow of control. The program is divided into a set of regions which correspond to states. The regions are interconnected by a net of GOTOs which correspond to state transitions. From the point of view of the straight-line and recursive PBMs, this net is liable to be totally unstructured, and unanalyzable. Looked at as a finite state automaton, however, it is very structured. For example, consider again a system which reads in requests from a user. Suppose that the system is structured so that requests can cause a switch from one subsystem to another. The overall system can be understood as a finite state automaton where the states are subsystems, and certain user requests can cause transitions between the states.

There are other types of GOTOs which could serve as models for useful PBMs. There are also other kinds of mechanisms, such as interrupt processing, which could be made into PBMs. The PBMs discussed in sections IV.1 and IV.2 are not intended to cover all reasonable methods of constructing programs. The discussion in this section shows that they, in fact, do not. However, they do cover a large percentage of the programs which are written. Adding new PBMs, though useful and an interesting area for further research, suffers from diminishing returns. Each additional PBM covers only a few additional programs. There are probably at most only a few dozen PBMs at the level of generality of the straight-line and recursive PBMs.

V. An Experiment

As mentioned above, it is possible to write a program which cannot be usefully analyzed in terms of the PBMs presented in the last chapter unless the program is radically transformed first. A question immediately arises. What percentage of those programs which programmers actually write can be usefully analyzed in terms of the PBMs without anything more than trivial transformations being done? The straight-line PBMs are similar to basic structured programming constructs, and can be used to analyze any straight-line program that does not have spaghetti-like control flow. It is reasonably clear that they are applicable to a large percentage of straight-line programs which programmers write.

The recursive PBMs are more novel, and it is not immediately clear how wide their range of applicability is. An experiment was performed in order to find out. 20% of the 220 programs in the IBM FORTRAN Scientific Subroutine Package (SSP) [49] were chosen at random and analyzed in terms of the PBMs by hand. The SSP was chosen because it is a large corpus of reasonably written programs which is an actual commercial product.

The 44 programs chosen contained 164 loops. (Other types of recursive programs are not expressible in FORTRAN.) These were analyzed as being built up by means of the PBM temporal composition out of 442 augmentations, 6 filters, and 186 terminations. The basic recursion role did not have to be used.

It was possible to analyze all of the loops with the PBMs. Configurations which cannot be analyzed (such as a loop with more than one entry point) did not occur in these loops. The sole difficulty was that one loop had multiple level error exits which branched outside of the loop from inside an inner loop. In order to analyze the error exits in terms of the PBMs above, they had to be looked at as being explicit exits from the inner loop and the outer loop. This may or may not be a good way to look at them. Looking at a more diverse class of FORTRAN programs, Allen and Cocke [1] found that 90% of the programs they looked at had reducible flow graphs which implies that they could be easily analyzed in terms of the recursive PBMs. Knuth's study of FORTRAN programs [58] also supports the basic idea that most loops are relatively simple, while only a few are really complex.

Another question which arises is, given that there is no limit to the complexity of the pieces which result from the analysis of a loop by PBMs, how often are the pieces simple? In the experiment being described here, it turned out that nearly 90% of the time, the pieces which resulted from PBM analysis were very simple. Of the 442 augmentations, 302 (68%) were easy to recognize as either a product, a sum, a max, a min, or a count. An additional 72 (16%) of the augmentations were conjunction augmentations which could not be pulled into another segment. They are all easy to understand because their recurrence relations do not reference prior values of the variable being defined. In total, 374 (85%) of the augmentations were easy to understand while only 31 (7%) were really difficult to understand. The remainder were of intermediate complexity. All 6 filters were simple comparisons with zero and therefore easily understood.

Of the 186 terminations, 165 (89%) were very simple to understand because they were simple comparisons with a fixed number, and the input which they tested was a simple sequence of numbers, most often {1,2,3, ...}. Due to the fact that their tests and inputs were so simple, it was trivial to determine the exact number of states associated with their inputs, and hence exactly when the loops they were in would terminate. Most of the 21 terminations that were more complex, were additional terminations in loops which had at least one simple termination. As a result of this, 162 (99%) of the 164 loops contained at least one simple termination, and therefore can be trivially shown to terminate.

In [77], T.W. Pratt reports an interesting experiment which supports the conclusion that a large percentage of loops are built up out of simple augmentations and terminations. He analyzed by hand all of the 170 loops in a compiler for PASCAL which was written in PASCAL. His goal was to investigate the control computation in these loops. The control computation is that part of the computation in a loop which controls its iteration and termination. From the point of view of PBMs, the control computation is equivalent to the terminations in a temporal composition and those augmentations which create temporal sequences of values tested by the terminations. He discovered that in 86% of the loops, the control computation had one of a few stereotyped forms. Stated in terms of PBMs his results show that in these loops the control computation can be analyzed as built up out of four different augmentations and two kinds of termination. The four augmentations are: enumerating the elements of a vector, enumerating the elements of a linked list, enumerating elements of a sequence of characters or symbols, and enumerating a sequence of integers by counting. The two terminations are: detecting that the last element of an aggregate structure (such as a list or vector) has been enumerated, and searching for an element which satisfies a particular property. Since all of these terminations and augmentations are easy to understand, Pratt's results show that the control computation for 86% of the loops can be straightforwardly analyzed and understood in terms of PBMs. His results are particularly interesting in that they are based on a language very different from FORTRAN, and a program very different from a mathematical *subroutine*.

In order to convey a better feeling for what the programs in the SSP, and their analyses in terms of PBMs, are like, the figures below show two examples. Figure V-1 shows the SSP program CONVT which copies a single precision matrix into a double precision matrix or vice versa. Figure V-2 shows a PBM structure diagram for this program. A PBM structure diagram represents the way a plan is built up by means of PBMS. The top level PBM in the plan appears in the upper left hand corner of the diagram followed by a colon. The names of the PBMs are abbreviated; "cond" standing for conditional, "pred" standing for predicate, etc. The role names associated with the PBM appear on separate lines indented so that they line up after the PBM name. These are also abbreviated; "act1" standing for action1, "aug1" standing for augmentation1, etc. Each role name is followed by a representation of the segment which fills the role.

If the role-filler is a terminal segment, then it is represented by a portion of the code for the program. For example, action1 of the initialization of CONVT is the operation * on line 32 of CONVT. It is represented by line 32 itself ("NM=N*M") in the PBM diagram. If a role-filler is not a terminal segment, but it is still contained entirely on one line of the program, then it is represented by the corresponding portion of the line of the program. For example, action2 of the initialization of CONVT is represented by "NM=((N+1)*N)/2" which is line 34 of CONVT. This action is itself an expression with three actions filled by the terminal segments +, *, and /. Augmentation1 of action1 of CONVT is represented by "DO 15 L=1," which is a portion of line 40 of CONVT. This augmentation counts up by 1 from 1. If a role-filler is spread over more than one line of the program, then a PBM structure diagram is used to describe the role-filler. For example, the initialization of CONVT is a conditional corresponding to lines 31 thru 36.

The PBM structure diagram does not explicitly show any of the control flow or data flow in the plan. The lines of code which appear in the diagram form a link between the diagram and the program. In order to strengthen this link, roles sometimes appear in a PBM structure diagram which would not actually appear in a plan. For example, action3 of the initialization of CONVT, "NM=N", would not have a segment corresponding to it in a plan for CONVT because there is no computation in action3. Rather, everything that happens in action3 would be expressed by the data flow in the plan. Action3 is included in the PBM structure diagram in order to make it easier to understand the relationship between the diagram and the program.

```
C      PURPOSE
C         CONVERT NUMBERS FROM SINGLE PRECISION TO DOUBLE PRECISION
C         OR FROM DOUBLE PRECISION TO SINGLE PRECISION.
C      DESCRIPTION OF PARAMETERS
C         N    - NUMBER OF ROWS IN MATRICES S AND D.
C         M    - NUMBER OF COLUMNS IN MATRICES S AND D.
C         MODE - CODE INDICATING TYPE OF CONVERSION
C                  1 - FROM SINGLE PRECISION TO DOUBLE PRECISION
C                  2 - FROM DOUBLE PRECISION TO SINGLE PRECISION
C         S    - IF MODE=1, THIS MATRIX CONTAINS SINGLE PRECISION
C                NUMBERS AS INPUT.  IF MODE=2, IT CONTAINS SINGLE
C                PRECISION NUMBERS AS OUTPUT.  THE SIZE OF MATRIX S
C                IS N BY M.
C         D    - IF MODE=1, THIS MATRIX CONTAINS DOUBLE PRECISION
C                NUMBERS AS OUTPUT.  IF MODE=2, IT CONTAINS DOUBLE
C                PRECISION NUMBERS AS INPUT.  THE SIZE OF MATRIX D IS
C                N BY M.
C         MS   - ONE DIGIT NUMBER FOR STORAGE MODE OF MATRIX
C                  0 - GENERAL
C                  1 - SYMMETRIC
C                  2 - DIAGONAL
C      METHOD
C         ACCORDING TO THE TYPE OF CONVERSION INDICATED IN MODE, THIS
C         SUBROUTINE COPIES NUMBERS FROM MATRIX S TO MATRIX D OR FROM
C         MATRIX D TO MATRIX S.
C
        SUBROUTINE CONVT (N,M,MODE,S,D,MS)
        DIMENSION S(1),D(1)
        DOUBLE PRECISION D
C          FIND STORAGE MODE OF MATRIX AND NUMBER OF DATA POINTS
        IF (MS-1) 2, 4, 6
      2 NM=N*M
        GO TO 8
      4 NM=((N+1)*N)/2
        GO TO 8
      6 NM=N
C          TEST TYPE OF CONVERSION
      8 IF (MODE-1) 10, 10, 20
C          SINGLE PRECISION TO DOUBLE PRECISION
     10 DO 15 L=1,NM
     15 D(L)=S(L)
        GO TO 30
C          DOUBLE PRECISION TO SINGLE PRECISION
     20 DO 25 L=1,NM
     25 S(L)=D(L)
     30 RETURN
        END
```

Figure V-1: The SSP program CONVT.

```
cond: init  cond: split IF (MS-1) 2,4,6
                  act1  NM=N*M
                  act2  NM=((N+1)*N)/2
                  act3  NM=N
                  join  at label 8
      split pred: IF (MODE-1) 10,10,20
      act1  temp comp: aug1  DO 15 L=1,
                       term1 DO 15     ,NM
                       aug2  D(L)=S(L)
      act2  temp comp: aug1  DO 25 L=1,
                       term1 DO 25     ,NM
                       aug2  S(L)=D(L)
      join  at label 30
```

Figure V-2: A PBM analysis of CONVT.

The PBM structure diagram indicates how CONVT can be analyzed in terms of PBMs. It specifies that the entire program is a conditional. As an initialization, CONVT computes the value NM. This computation is itself a conditional. NM is computed in one of three ways based on the parameter MS. The top level conditional is controlled by a split based on the value of the parameter MODE. Action1 is a temporal composition which has three parts. Augmentation1 enumerates integers starting with one. Termination1 truncates this sequence at NM. Augmentation2 copies the indicated elements from vector S to vector D. All three of these role-fillers are simple to understand. Action2 is very similar. The plan for this program is discussed in much more detail in chapters VI and VII.

```
C      PURPOSE
C         INTEGRATES A FIRST ORDER DIFFERENTIAL EQUATION
C         DY/DX=FUN(X,Y) UP TO A SPECIFIED FINAL VALUE
C      DESCRIPTION OF PARAMETERS
C         FUN -USER-SUPPLIED FUNCTION SUBPROGRAM WITH
C              ARGUMENTS X,Y WHICH GIVES DY/DX
C         HI  -THE STEP SIZE
C         XI  -INITIAL VALUE OF X
C         YI  -INITIAL VALUE OF Y WHERE YI=F(XI)
C         XF  -FINAL VALUE OF X
C         YF  -FINAL VALUE OF Y
C         ANSX-RESULTANT VALUE OF X
C         ANSY-RESULTANT VALUE OF Y
C              EITHER ANSX=XF OR ANSY=YF DEPENDING
C              ON WHICH IS REACHED FIRST
C         IER -ERROR CODE
C              IER=0 NO ERROR
C              IER=1 STEP SIZE IS ZERO
C      REMARKS
C         IF XI IS GREATER THAN OR EQUAL TO XF, ANSX=XI AND ANSY=YI
C         IF HI IS ZERO, IER=1, ANSX=XI, AND ANSY=0.0
C      METHOD
C         USES FOURTH ORDER RUNGE-KUTTA INTEGRATION
C         PROCESS ON A RECURSIVE BASIS.  PROCESS IS
C         TERMINATED AND FINAL VALUE ADJUSTED WHEN
C         EITHER XF OF YF IS REACHED
C
       SUBROUTINE RK1(FUN,HI,XI,YI,XF,YF,ANSX,ANSY,IER)
C      IF XF IS LESS THAN OR EQUAL TO XI, RETURN (XI,YI)
       IER=0
       IF (XF-XI) 11,11,12
    11 ANSX=XI
       ANSY=YI
       RETURN
```

```
C     TEST INTERVAL VALUE
   12 H=HI
      IF (HI) 16,14,20
   14 IER=1
      ANSX=XI
      ANSY=0.0
      RETURN
   16 H=-HI
C     SET XN=INITIAL X, YN=INITIAL Y
   20 XN=XI
      YN=YI
C     INTEGRATE ONE TIME STEP
      HNEW=H
      JUMP=1
      GOTO 170
   25 XN1=XX
      YN1=YY
C     COMPARE XN1 (=X(N+1)) TO X FINAL AND BRANCH ACCORDINGLY
      IF (XN1-XF) 50,30,40
C     XN1=XF, RETURN (XF,YN1) AS ANSWER
   30 ANSX=XF
      ANSY=YN1
      GOTO 160
C     XN1 GREATER THAN X FINAL, SET NEW STEP SIZE AND
C     INTEGRATE ONE STEP, RETURN RESULTS AS ANSWER
   40 HNEW=XF-XN
      JUMP=2
      GOTO 170
   45 ANSX=XX
      ANSY=YY
      GOTO 160
C     XN1 LESS THAN X FINAL, CHECK IF (YN,YN1) SPAN Y FINAL
   50 IF ((YN1-YF)*(YF-YN)) 60,70,110
C     (YN1,YN) DOES NOT SPAN YF, SET (XN,YN)=(XN1,YN1), REPEAT
   60 YN=YN1
      XN=XN1
      GOTO 170
C     EITHER YN OR YN1 =YF. CHECK WHICH AND RETURN PROPER (X,Y)
   70 IF (YN1-YF) 80,100,80
   80 ANSY=YN
      ANSX=XN
      GOTO 160
  100 ANSY=YN1
      ANSX=XN1
      GOTO 160
C     (YN,YN1) SPANS YF. TRY TO FIND X VALUE ASSOCIATED WITH YF
  110 DO 140 I=1,10
C     INTERPOLATE TO FIND NEW TIME STEP AND INTEGRATE ONE STEP
C     TRY TEN INTERPOLATIONS AT MOST
      HNEW=((YF-YN)/(YN1-YN))*(XN1-XN)
      JUMP=3
      GOTO 170
  115 XNEW=XX
      YNEW=YY
C     COMPARE COMPUTED Y VALUE WITH YF AND BRANCH
      IF (YNEW-YF) 120,150,130
C     ADVANCE, YF IS BETWEEN YNEW AND YN1
  120 YN=YNEW
      XN=XNEW
      GOTO 140
C     ADVANCE, YF IS BETWEEN YN AND YNEW
  130 YN1=YNEW
```

```
 97           XN1=XNEW
 98       140 CONTINUE
 99     C     RETURN (XNEW,YF) AS ANSWER
100       150 ANSX=XNEW
101           ANSY=YF
102       160 RETURN
103     C
104       170 H2=HNEW/2.0
105           T1=HNEW*FUN(XN,YN)
106           T2=HNEW*FUN(XN+H2,YN+T1/2.0)
107           T3=HNEW*FUN(XN+H2,YN+T2/2.0)
108           T4=HNEW*FUN(XN+HNEW,YN+T3)
109           YY=YN+(T1+2.0*T2+2.0*T3+T4)/6.0
110           XX=XN+HNEW
111           GOTO (25,45,115),JUMP
112           END
```

Figure V-3: The SSP program RK1.

```
cond: init  IER=0
      split pred: split1 IF(XF-XI)11,11,12
                  split2 IF(HI) ,14,
      act1  ANSX=XI, ANSY=YI
      act2  IER=1, ANSX=XI, ANSY=0.0
      act3  cond: init  cond: init  H=HI
                                split IF(HI)16,,20
                                act1  H=-HI
                        XN=XI, YN=YI, HNEW=H
                  split temp comp: aug1  XN1=XN+HNEW
                                   term1 IF(XN1-XF)50,,
                                   aug2  YN1=integrate(HNEW,XN,YN)
                                   term2 IF((YN1-YF)*(YF-YN))60,,
                  act1  cond: split IF(XN1-XF) ,30,40
                              act1  ANSX=XF, ANSY=YN1
                              act2  HNEW=XF-YN, ANSX=XN+HNEW,
                                    ANSY=integrate(HNEW,XN,YN)
                              join  at label 160
                  act2  cond: split pred: split1 IF((YN1-YF)*(YF-YN)) ,70,110
                                          split2 IF(YN1-YF)80,100,80
                              act1  ANSY=YN, ANSX=XN
                              act2  ANSY=YN1, ANSX=XN1
                              act3  temp comp: aug1  DO 140 I=1,
                                               term1 DO 140    ,10
                                               aug2  HNEW=((YF-YN)/(YN1-YN))*(XN1-XN)
                                                     XNEW=XN+HNEW
                                                     YNEW=integrate(HNEW,XN,YN)
                                                     cond: split IF(YNEW-YF)120,,130
                                                           act1  XN=XNEW, YN=YNEW
                                                           act2  XN1=XNEW, YN1=YNEW
                                                           join  at label 140
                                               term2 IF (YNEW-YF)  ,150,
                              join  at label 160
                  join  at RETURN
      join  at RETURN
```

Figure V-4: A PBM analysis of RK1.

The SSP program RK1 (see Figure V-3) which performs numerical integration is more complex than the program CONVT. A considerable syntactic complexity in the program is the use of a computed goto (line 111) in order to make lines 104 thru 110 act like a subroutine. When analyzing the program this can be dealt with by either making these lines into a real subroutine, or, as was done here, by simply inserting copies of the lines in the places where they are used (namely at labels 25, 44, and 115). (Note that if the computed goto is viewed as a fundamental part of the program, and not just a syntactic abbreviation, then the program's control flow is irreducible and the program cannot be analyzed.)

The PBM structure diagram for RK1 in Figure V-4 has been abbreviated somewhat in order to make it smaller. Instances of the PBMs expression, conjunction, and composition have been represented by a simple list of their actions. For example, action2 of the top level conditional is a conjunction. In addition, "integrate(HNEW,XN,YN)" is used as an abbreviation for the expression on lines 104 thru 109.

The top level PBM in RK1 is a conditional. If the final X value XF is not greater than the initial X value XI, then RK1 returns the initial values as the result (action1). If the step size HI is zero, then RK1 returns setting an error flag (action2). Otherwise, the program performs numerical integration (action3). Action3 is a conditional reflecting the fact that there are two different operations which RK1 can perform. Given XI and XF, it can compute the integral of FUN from XI to XF. Alternatively, given XI and YF, it can compute XF such that the integral of FUN from XI to XF equals YF. Action3 has an initialization which computes the absolute value of the initial step size HI. (It is interesting to note that the split of this conditional, and split2 of the split of RK1 are both implemented by means of the same arithmetic IF in RK1.)

The split of action3 is a temporal composition with two terminations. Augmentation1 of this loop is a counter which starts at XI and counts up by HNEW, the integrating step size. Termination1 stops the loop if the X value reaches XF. Augmentation1 and termination1 are both easy to understand, and it is clear that the loop will terminate. Augmentation2 performs the integration. It is an example of an augmentation which is hard to understand except as a recurrence relation. Termination2 halts the loop if the value of Y crosses over YF. By itself this termination is of moderate complexity. However, Augmentation2 must also be understood in order to understand when termination2 will cause the loop to terminate.

If the loop terminates because XF is reached, then action1 of action3 of RK1 is performed. This is a conditional which is based on whether the X value exactly reached XF or went beyond XF. If XF was reached exactly, then the current values of X and Y are returned. Otherwise, one final step of integration is performed in order to get the exact values desired.

If the loop terminates because the Y values crossed over YF then action2 of action3 of RK1 is performed. This action is a three way conditional. If, either the last value of Y or the current value of Y exactly matches YF, then the corresponding results are returned (action1 and action2). Otherwise, action3 of action2 of action3 of RK1 performs a search looking for the value of X corresponding to YF. Augmentation1 of this loop counts up from 1. Termination1 stops the loop when 10 is reached. These two role-fillers have nothing to do with the searching, they merely ensure that the loop will terminate. Augmentation2 does the actual searching. It is a very complex composition augmentation. It would only be understood by the methods described here if the particular type of searching being done could be recognized as a special case. Termination2 halts the loop if the correct value is found. This termination is very easy to understand by itself. However, due to the complexity of augmentation2, their combined actions are not easy to understand.

The analysis of RK1 in terms of PBMs makes it easier to understand the program because it breaks the program up into pieces which are combined in a straightforward manner. Most of the pieces are themselves easy to understand while some of the pieces are quite complex. It is important to note that the PBM analysis separates out the difficult parts of RK1 so that the parts which are simple can be easily understood and so that more powerful methods can be applied specifically to the more difficult parts. As written, RK1 is not very easy to understand because its logical structure is not very clear. The PBM analysis makes the structure of the program clearer.

VI. An Automatic Analysis System

The author has implemented a system which automatically analyzes programs in terms of the PBMs. The system operates in two phases. A translation phase reads in a program and converts it to a surface plan for the program. An analysis phase then analyzes the program by looking at the surface plan. The subsections below describe each part of the analysis system in detail. They also show a detailed example of a program being analyzed.

VI.1 The Translation Phase

The translation process operates in two steps. The first step reads in a FORTRAN subroutine and converts it into an executable LISP program. The program is converted into LISP so that it can be more conveniently worked on by the analysis system, which is itself written in LISP.

The translator accepts programs written in a subset of FORTRAN IV [48]. The subset corresponds to the language features actually used in the subroutines in the IBM SSP [49] which has been the major test bed for the translation phase of the analysis system. Most of the features omitted are of no theoretical interest because they are just abbreviations for things which are expressible in the subset. They include: logical, character, and complex data types; logical and relational operators, the assigned goto statement, the logical if statement, and statement functions. Some of the other features omitted are of fundamental importance, and make the analysis task significantly easier. They include: the common statement, the equivalence statement, and all input/output statements. It is important to note that in keeping with the programs in the SSP, it is assumed that there are absolutely no side-effects other than assignment to variables and to elements of arrays. It is also assumed that two variables which have different names can never refer to the same, or overlapping, memory locations.

The output of step one of the translator is a MACLISP [70] program. However, it does not use any of the standard MACLISP functions. Rather, the result is stated in terms of a set of functions which implement the semantics of FORTRAN and which maintain the syntax of FORTRAN as closely as possible. The specialized functions have all been implemented so that the LISP form of a FORTRAN program can actually be executed, all be it slowly.

```
      SUBROUTINE CONVT (N,M,MODE,S,D,MS)
      DIMENSION S(1),D(1)
      DOUBLE PRECISION D
      IF (MS-1) 2, 4, 6
    2 NM=N*M
      GO TO 8
    4 NM=((N+1)*N)/2
      GO TO 8
    6 NM=N
    8 IF (MODE-1) 10, 10, 20
   10 DO 15 L=1,NM
   15 D(L)=S(L)
      GO TO 30
   20 DO 25 L=1,NM
   25 S(L)=D(L)
   30 RETURN
      END
```

Figure VI-1: The program CONVT.

```
(_SUBROUTINE CONVT
               (N M MODE S D MS)
               (_COMMENT (SUB_VAR_TYPES (S _R) (L _I) (MODE _I) (NM _I)
                                         (N _I) (M _I) (_T1 _I) (MS _I) (D _D)))
               (_COMMENT (SUB_VAR_CTYPES (D _A 1) (S _A 1)))
               (_= _T1 (_-I MS (_I 1)))
               (_IFI (_LT _T1) _2)
               (_IFI (_EQ _T1) _4)
               (_GOTO _6)
 _2            (_= NM (_*I N M))
               (_GOTO _8)
 _4            (_= NM (_//I (_*I (_+I N (_I 1)) N) (_I 2)))
               (_GOTO _8)
 _6            (_= NM N)
 _8            (_IFI (_GT (_-I MODE (_I 1))) _20)
               (_GOTO _10)
 _10           (_= L (_I 1))
 _01 _15       (_A= D (1) L (_RTOD (_AREF S (1) L)))
               (_= L (_+I L (_I 1)))
               (_IFI (_GT (_-I L NM)) _02)
               (_GOTO _01)
 _02           (_GOTO _30)
 _20           (_= L (_I 1))
 _03 _25       (_A= S (1) L (_DTOR (_AREF D (1) L)))
               (_= L (_+I L (_I 1)))
               (_IFI (_GT (_-I L NM)) _04)
               (_GOTO _03)
 _04 _30       (_GOTO _RETURN))
```

Figure VI-2: The LISP form of CONVT.

The LISP version of CONVT is shown in Figure VI-2. (It should be noted that all of the figures in Chapter V and Chapter VI are adapted directly from actual output of the analysis system being described.) The FORTRAN version appears in Figure VI-1. The translation is performed by parsing the original program by means of a grammar for FORTRAN, and then reexpanding it with terminal nodes appropriate to the LISP form. The only complexity involved comes from the fact that some information which is implicit and/or global in the FORTRAN version is made explicit and local in the LISP version.

The line beginning "(_COMMENT (SUB_VAR_TYPES" in the LISP version gives the type of every variable in the program. The type is specified by a code: _I indicates an integer variable, _R indicates a real variable, and _D indicates a double precision variable. For example S is a real number, L is an integer, and D is a double precision real number. The type information is gathered from the declarations, and the variable names. The next line specifies additional type information. Here it says that S and D are one dimensional arrays.

Several changes have been made in the program itself. The *arithmetic IF statements* have been *macro expanded* in terms of simple conditional branching statements. The DO statements have also been replaced by macro expansions. In a similar vein, RETURN statements are replaced by a GOTO to the special label _RETURN. These macro expansions are done in order to simplify the constructs which the second step of translation has to deal with.

Each literal number is replaced by a function call which returns the appropriate value and type. For example, "1" becomes "(_I 1)". Each generic operator in the source (such as +) has been replaced by a specific operation which accepts arguments of a known type (such as _+I which adds integers and returns an integer). As part of this, any implicit conversions are made explicit. For example, the function _RTOD has been inserted in the line labeled 15 in order to make it explicit that a conversion from real to double precision is taking place. Array references are made explicit by means of the function _AREF which takes the declared bounds of the array as an argument. One of the reasons behind these last two transformations is the desire to make all of the information needed for executing the program completely local.

The second translation step converts the LISP form into a surface plan. It operates in seven stages. In the stages, the interrelationship between statements in the program is handled differently from the interrelationship between operations within a statement. This is done, because in the semantics for FORTRAN, what can happen in a statement is completely different from what can happen between statements. This distinction does not exist in a language like LISP.

In the first stage of the second step of translation, a segment is created for the program as a whole. The evolving plan for the program is stored in the form of assertions in a relational data base. Inputs are added to the segment corresponding to the arguments of the subroutine and pre-conditions are added based on the types of the arguments.

The second stage symbolically evaluates the subroutine on a statement by statement basis, treating the statements as indivisible entities, in order to determine what the inter-statement control flow is. After this stage, GOTO statements and labels are ignored. Their only function is to specify control flow and they do not appear in the surface plan because the control flow is represented explicitly by control flow arcs in the plan. It should be noted, that looping in the program appears as looping in the control flow in the surface plan after translation. This looping is not replaced by recursion until the analysis phase.

The third stage analyzes the result of the second stage in order to group straight-line portions of the program into "control environments" which are used by the later stages. Figure VI-3 shows the situation after the first three stages of translation. In the figure, the inter-statement control flow is represented by control flow arcs, each statement is represented by the FORTRAN code corresponding to it, and a circle is used to indicate each control environment which contains more than one statement.

The fourth stage constructs a complete surface plan for each statement in isolation. This stage takes advantage of the fact that within a statement, FORTRAN is completely applicative. The data flow between parts of a statement is implemented solely in terms of nesting of expressions. A statement can receive values from variables, but a variable cannot be used to communicate a value between two parts of the same statement. The control flow in a statement is straight-line, controlled by the left to right order of argument evaluation.

The fourth stage works by symbolically evaluating each statement. A terminal segment is created for each operator in the statement. This is done by copying a template for the operator which specifies its inputs, outputs, pre-conditions, and post-conditions. The translator has no knowledge of the basic operators other than what is in these templates. Terminal segments are also constructed for each constant in the statement. The data flow and control flow is constructed based on the nested structure of the statement. Any references to variables are left dangling. They are attached to the other statements in a later stage. As the surface plan is built up, links are entered into the data base which indicate what parts of the code for the program give rise to each segment, each control flow, and each data flow in the plan.

Figure VI-4 shows three examples of statements transformed by this stage. Assignment statements and variables do not appear in the final plan because their only function is to specify

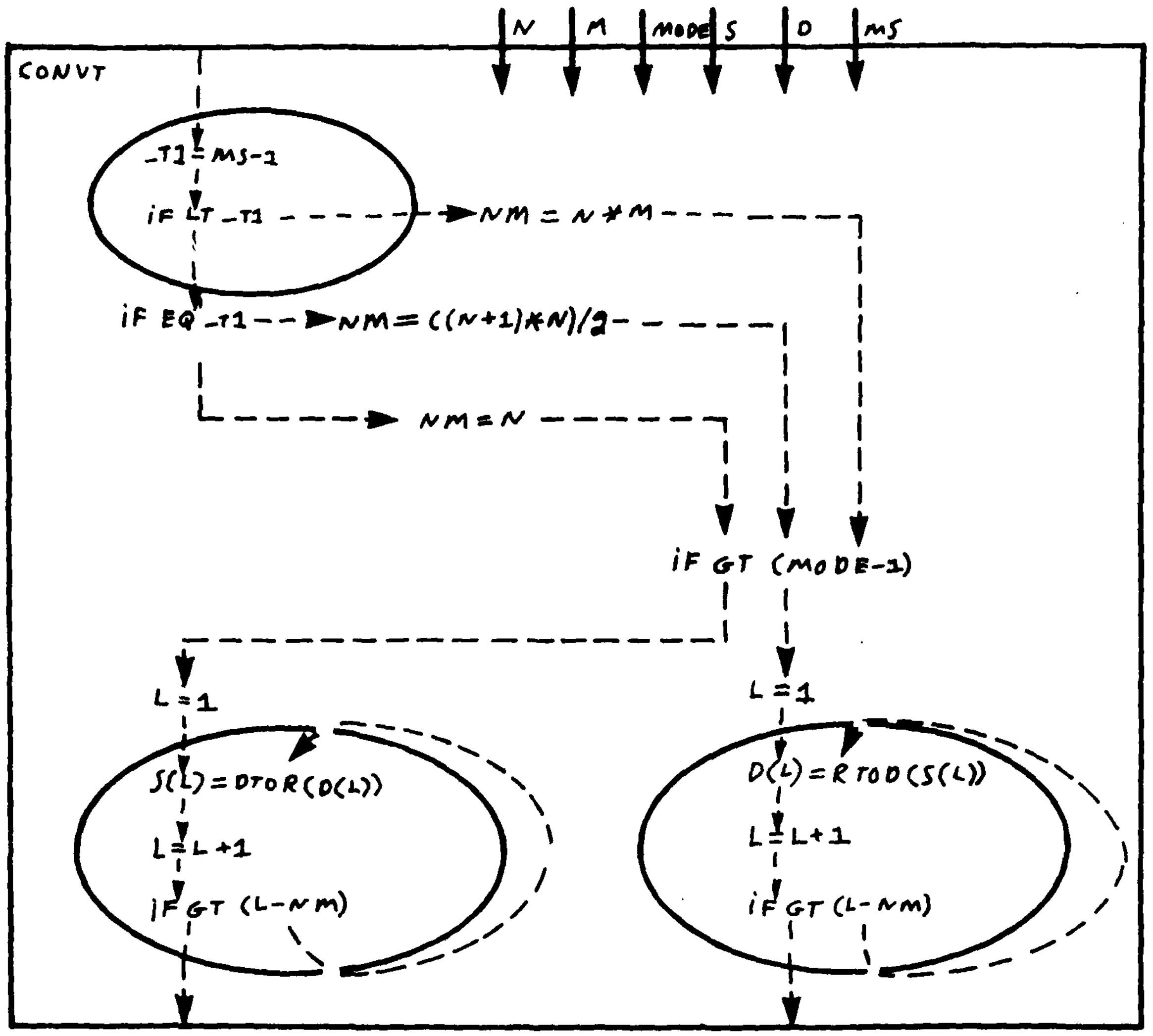


Figure VI-3: Inter-statement control flow and control environments.

data flow, and data flow is explicitly represented in the plan by data flow arcs. Looking at the figure, it can be seen that nothing remains of the assignment operation in the first example. Rather, the *value is just carried off in* the variable NM. The third example shows that something does remain for this assignment because, it is an array assignment and the operator A= is used to represent the action of side-effecting the array by storing a new value in it. The side-effected array is an explicit output of this operator.

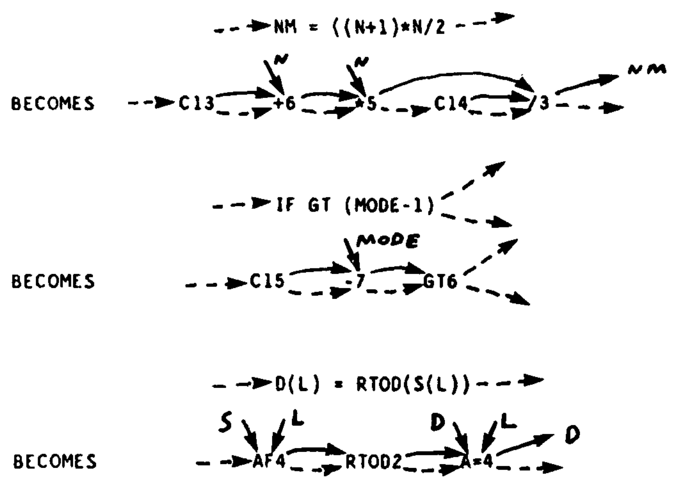


Figure VI-4: Intra-statement control flow and data flow.

The fifth stage connects up the control flow between statements to the appropriate terminal segments. A control flow from statement A to statement B is connected up between the last (in order of control flow) terminal segment in A and the first terminal segment in B.

The primary purpose of the sixth stage is to determine what joins must be put into the surface plan. A join is needed whenever more than one control flow terminates on the same side of the same case of the same segment. The major difficulty is in determining which data flows have to pass through the join, and which do not. The only data flows which have to pass through a join are ones which have different sources corresponding to different control flow paths which lead to the join. For example, referring to Figure VI-1, there must be a join corresponding to the label 8 after the three part conditional at the beginning of CONVT. Data flow corresponding to the variable NM must pass through this join because it comes from either line 2, 4, or 6, depending on what control flow path is taken to the join. Data flow corresponding to the variable MODE does not have to go through the join because it has the same source (an input argument) no matter what control flow path is followed to the join.

The sixth stage processes the program on a control environment by control environment basis. It first determines what variables are used as inputs by each control environment and which variables are assigned to by each control environment. Transitive closures are then computed in order to determine every variable which must be available before executing a given control environment, and all of the variables available after executing the control environment. The stage then examines each place where a join is needed and checks to see which needed variables have different sources in the different control environments feeding into the join. It then creates a join with the appropriate inputs and outputs.

The seventh and final stage puts in all of the inter-statement data flow. It does this by symbolically executing the program remembering the source associated with each variable. It takes advantage of the fact that once the joins have been put in, each variable only has one source associated with it. This is one of the simplifications which the presence of joins allows.

The seventh stage uses the information computed in the sixth stage in order to determine what outputs the subroutine has. Each of the arguments to the subroutine could be treated as an output.

However, the only arguments which are treated as outputs in the plan are those which are capable of being modified in the subroutine. In CONVT the only arguments treated as outputs are S and D. Figure VI-5 shows the completed urface plan for CONVT. (Some of the data flow corresponding to S and D has been abbreviated in order to simplify the figure.)

The transformation phase is simplified by the fact that it proceeds without any specific knowledge of what the primitive functions in the language do. Its operation is based solely on knowledge of the constructs which implement control flow (such as GOTO and IF), and constructs which implement data flow (such as variables, and assignment). Its success depends upon the assumption that the data flow and control flow can actually be determined without knowing what computations are being performed. Unfortunately, there are at least two situations in which this is not the case.

The first problem is with side-effects. In general, it is not possible to determine what data flow is mediated by a side-effect without understanding the operations which are affecting the relevant data objects. This problem is avoided in the current implementation because programs with side-effects are not allowed.

The second problem comes from the fact that it is possible for the splits in a program to interact in such a way that not all control flow paths in the program are accessible. If this kind of situation can be detected, then the control flow and, as a result, the data flow can be simplified. Unfortunately, this cannot be detected without knowing what tests the splits are performing. The current implementation is forced to make the pessimistic assumption that all control flow paths are accessible.

An example which illustrates this problem is the computed GOTO on line 111 of the program RK1 in Figure V-3. The purpose of this computed GOTO is to make lines 104-110 act like a subroutine. This pseudo-subroutine is called from lines 49, 62, 71, and 86. The variable JUMP which is set on lines 48, 61, and 85 is used to control the computed GOTO. When the pseudo-subroutine is called from lines 49 or 71, it returns to line 50; when it is called from line 62, it returns to line 63; and when it is called from line 86, it returns to line 87. Because the current translator does not understand anything about what is being computed, it cannot detect this relationship between the place the computed GOTO branches to and the place the pseudo-subroutine was called from. It has to make the pessimistic assumption that all three destinations can be reached from each of the four calling points. This produces twelve control flow paths, only four of which, are accessible in the program. The extra control flow paths lead to a multiplicity of excess data flow arcs. In order to get an appropriate surface plan for RK1 with the current translator, the computed GOTO has to be removed, and the pseudo-subroutine has to be converted into a real subroutine which is called where needed.

An improved translator is currently being constructed which deals with situations (such as the above) where the use of flags makes the control flow in a program appear anomalously complex. During the stage where the new translator determines what the basic inter-statement control flow is, it keeps track of the values of flag variables which are assigned constant values. When a join point is encountered, it checks to see whether there are any flags which are known to have different values on different control flow paths entering the join point. If there are, then the code after the join is duplicated so that there is a copy of the code corresponding to each of the different states of knowledge about the flag variable. When a split is encountered which compares one of the flag variables with a constant, then the test is performed in order to try and determine which case will

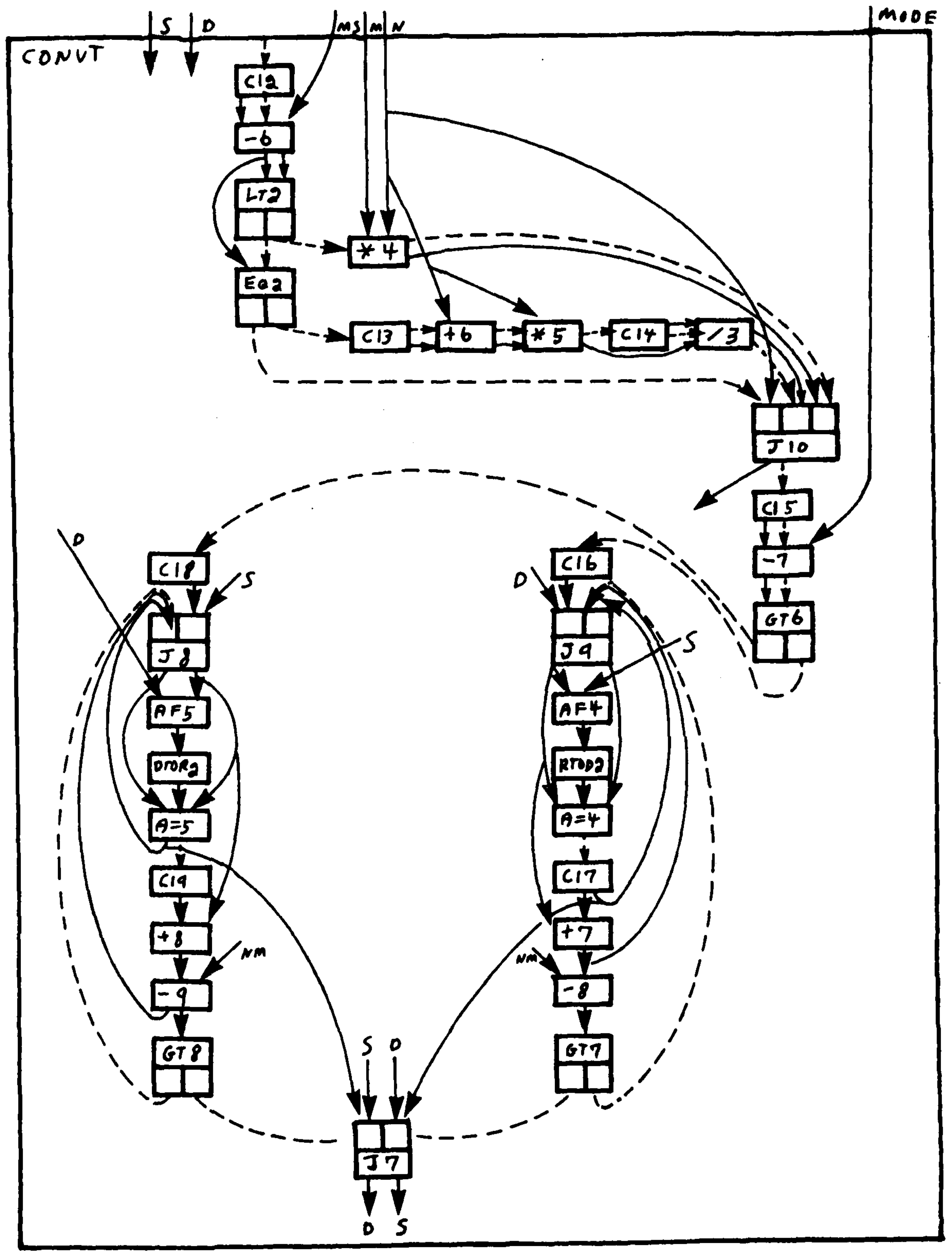


Figure VI-5: The surface plan for CONVT.

be activated. (Note that the code duplication at joins guarantees that there will only be a single value for each flag variable.) If the case which will execute can be determined, then the split itself is deleted and replaced by a direct branch since it no longer corresponds to a real choice. The sections of code accessible from the non-selected cases of the choice will also be deleted unless they are accessible in some other way. This eliminates code which would otherwise be unnecessarily duplicated.

Consider how this process would work for the program RK1. The variable JUMP is assigned three different constant values (1, 2, and 3). On each of the four control flow paths entering the pseudo-subroutine at line 104, JUMP is known to have just one of these values. This causes the pseudo-subroutine to be duplicated three times. Following this copying, the branch to be taken by the computed GOTO on line 111 in each of the three copies is clear, and therefore the computed GOTO is replaced by a simple GOTO in each copy. Once this is done, nothing in the program uses the value of the flag JUMP. Therefore, the lines where it is assigned to can also be deleted. As a result of all this, the pseudo-subroutine is replaced by three distinct copies, and all references to the flag JUMP are eliminated from the program.

VI.2 The Analysis Phase

It should be noted that the analysis phase operates on surface plans, and does not have any dependence on FORTRAN. Charles Rich [79] has written a translator which constructs surface plans for LISP programs. The analysis procedure described below operates just as well on these surface plans as on the surface plans derived from FORTRAN programs.

VI.2.1 Step 1: Basic Segmentation

The first step of the analysis phase analyzes the surface plan in terms of the four PBMs composition, predicate, conditional, and single self recursion. It does this bottom up by recognizing groups of segments which can be grouped together in accordance with one of these PBMs. Figure VI-6 shows the basic patterns which are recognized. The first step of analysis uses these patterns in order to analyze a surface plan as follows. The top level of the plan is searched in order to see whether a set of segments in it can be found which matches one of the patterns. If such a set of segments is found, then it is grouped together into a new segment. This is done by making all of the segments in the set subsegments of a new segment. The new segment is given ports and cases so that all of the data flow and control flow to the subsegments can be routed through the new segment. Once a set of segments is grouped together, they are no longer eligible to match a pattern, because they are no longer at the top level in the plan. However, the newly created supersegment is eligible to match patterns. Once the grouping is completed, the plan is again searched to look for another match.

The matching process continues until either, a pattern is found which matches all of the remaining top level segments in the plan, in which case the analysis is complete, or a situation is reached where no pattern match can be found, in which case the analysis has failed. The later will happen with any of the unanalyzable programs discussed above. The first stage of analysis is guaranteed to terminate because the application of each pattern strictly reduces the number of top level segments.

The first pattern looked for is the pattern in part A of Figure VI-6. This pattern corresponds to

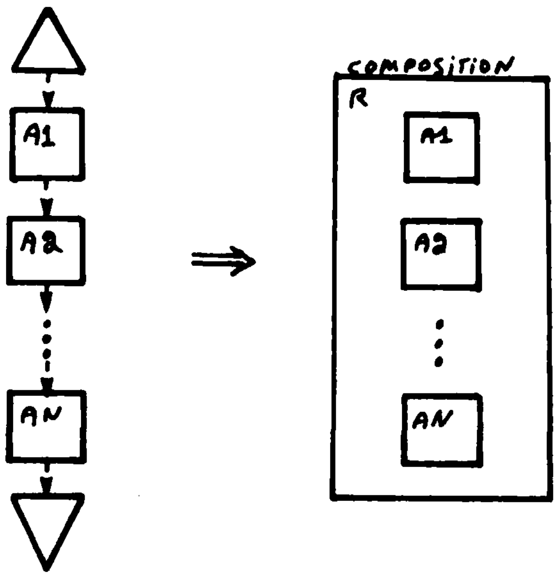


Figure VI-6: The simple patterns recognized (Part A).

a chain of two or more straight-line segments connected by control flow. It represents a maximal chain, in that the chain cannot be extended to include either the segment before the first one, or the segment after the last one. If such a pattern is found, the segments are grouped together into a composition. Note that when this is done, the control flow between the subsegments is deleted. This is done in order to canonicalize the representation of a composition. In a composition, the only restrictions on execution order which are necessary are expressed by the data flow arcs. The resulting composition is not further analyzed until the third step of analysis. As an efficiency measure, the analyzer takes advantage of the control environments determined in the translation phase by grouping the straight-line segments in each control environment into a composition before doing anything else. This reduces the number of top level segments, thereby speeding up the search for pattern matches.

If no compositions are found, then the analyzer searches for the three patterns in part B of Figure VI-6. The first of these patterns matches a split which is preceded by a straight-line segment which has all of its data flow going directly to the split. These two segments can be grouped together into a predicate with the straight-line segment as the initialization.

The second pattern locates primitive conditionals. The circles in the pattern correspond to optional straight-line segments. As a result, there are four basic types of conditionals represented by the pattern. The join is not actually required to be a simple binary join. It can be part of a larger join which will be broken apart when the grouping is done. At the current time, the analyzer assumes that the splits it sees will be binary.

The third pattern corresponds to a one exit loop. When such a loop is found, it is converted into a single self recursion. Note that the join which appears in the single self recursion is not logically equivalent to the join in the loop. The join in the loop is associated with starting the loop up, and is equivalent to the input side of the body of the single self recursion. The join in the single self recursion is associated with the termination of the recursion and is not needed in the loop. A loop is the only kind of recursion which is looked for with a pattern. Recursions which are written as

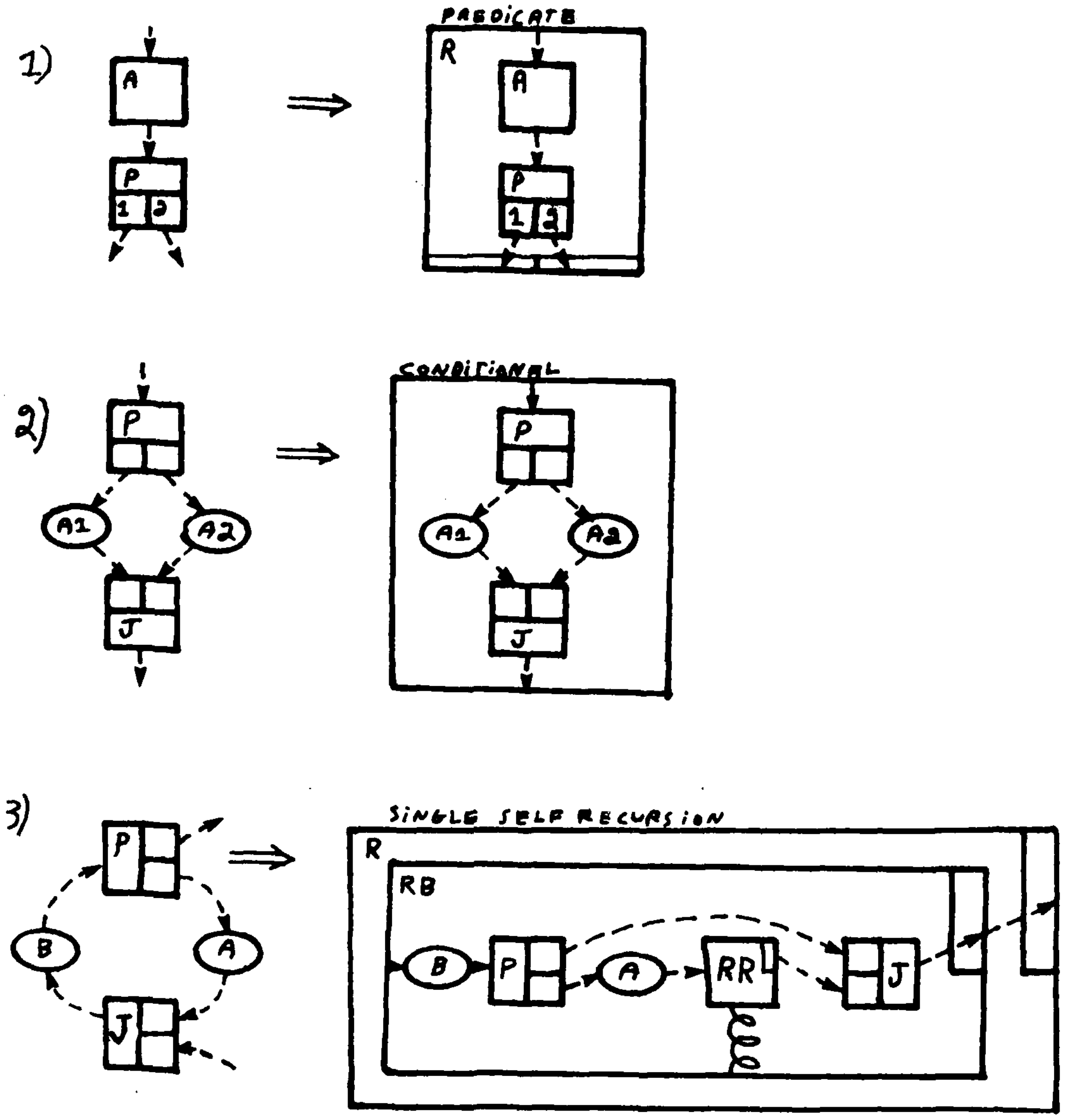


Figure VI-6: The simple patterns recognized (Part B).

recursive function calls, are already in recursive form in the surface plan. The subroutine boundaries in the code specify what the segmentation corresponding to the single self recursion is. As a result, the segmentation does not have to be recognized.

The four patterns described above are sufficient for analyzing a program like CONVT. Figure VI-7 shows the segmentation which results. To simplify the figure, the data flow and most of the control flow has been omitted. This figure can best be understood in comparison with Figure VI-5. Initially, based on control environments, C12 and -6 are grouped together into a composition, as are C13 thru /3, C15 and -7, AF4 thru -8, and AF5 thru -9. This grouping reduces the number of top level segments from 33 to 17. After this, EQ2, the composition of C13 thru /2, and part of join J10 are recognized as a conditional. Once this is grouped up, it is possible to recognize that LT2, *4, the conditional just formed, and the remainder of J10 can be grouped as

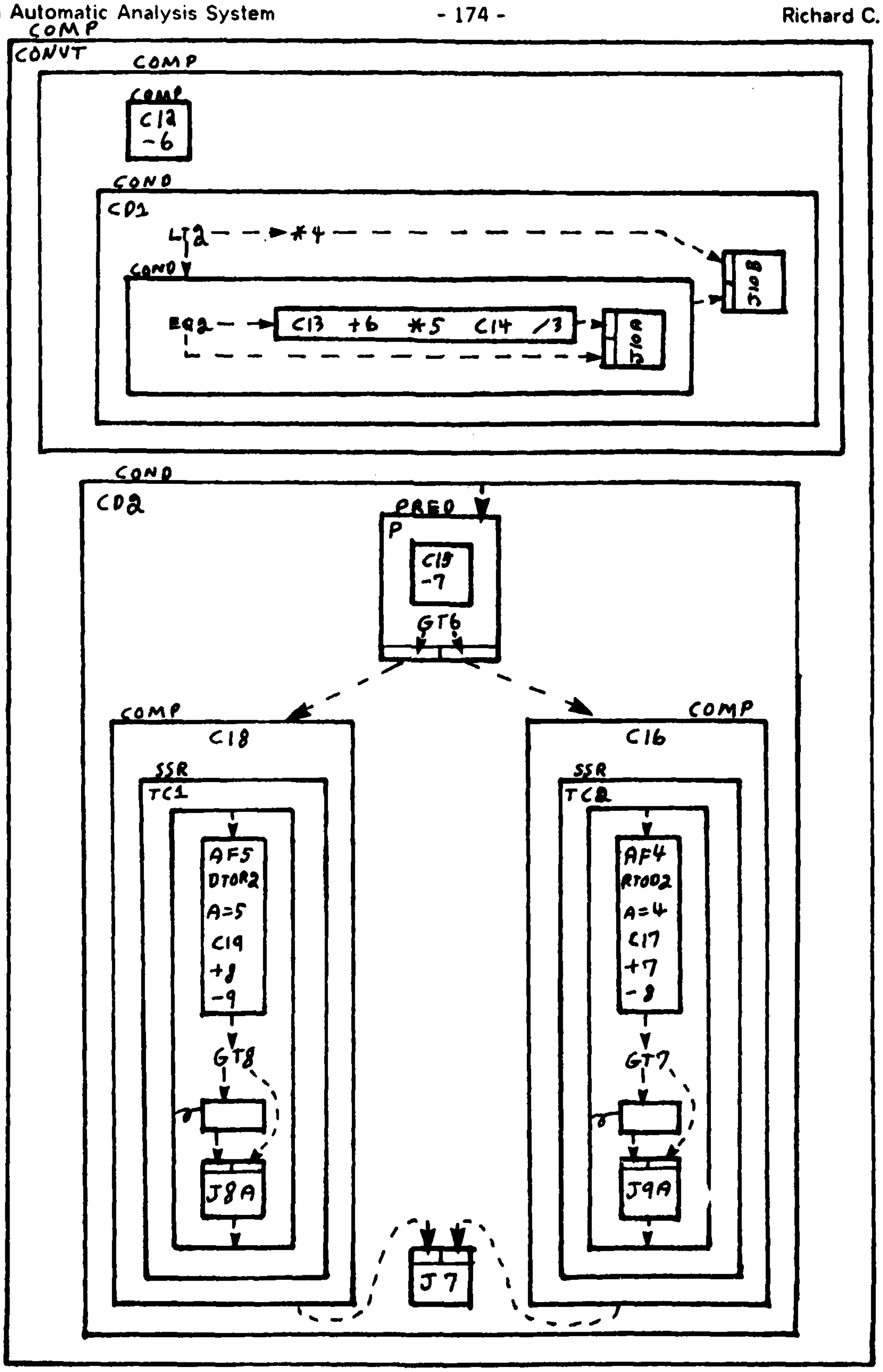


Figure VI-7: Initial segmentation of CONVT.

another conditional. The composition of C15 and -7 is combined with GT6 as a predicate.

Join J9, the composition of AF4 thru -8, and GT7 are recognized as a single self recursion. The result is then combined with C16 as a composition. The other loop is treated similarly. This allows the predicate, the two single self recursions, and J7 to be combined as a conditional. As a final step, CONVT is recognized as a composition of its two top level segments.

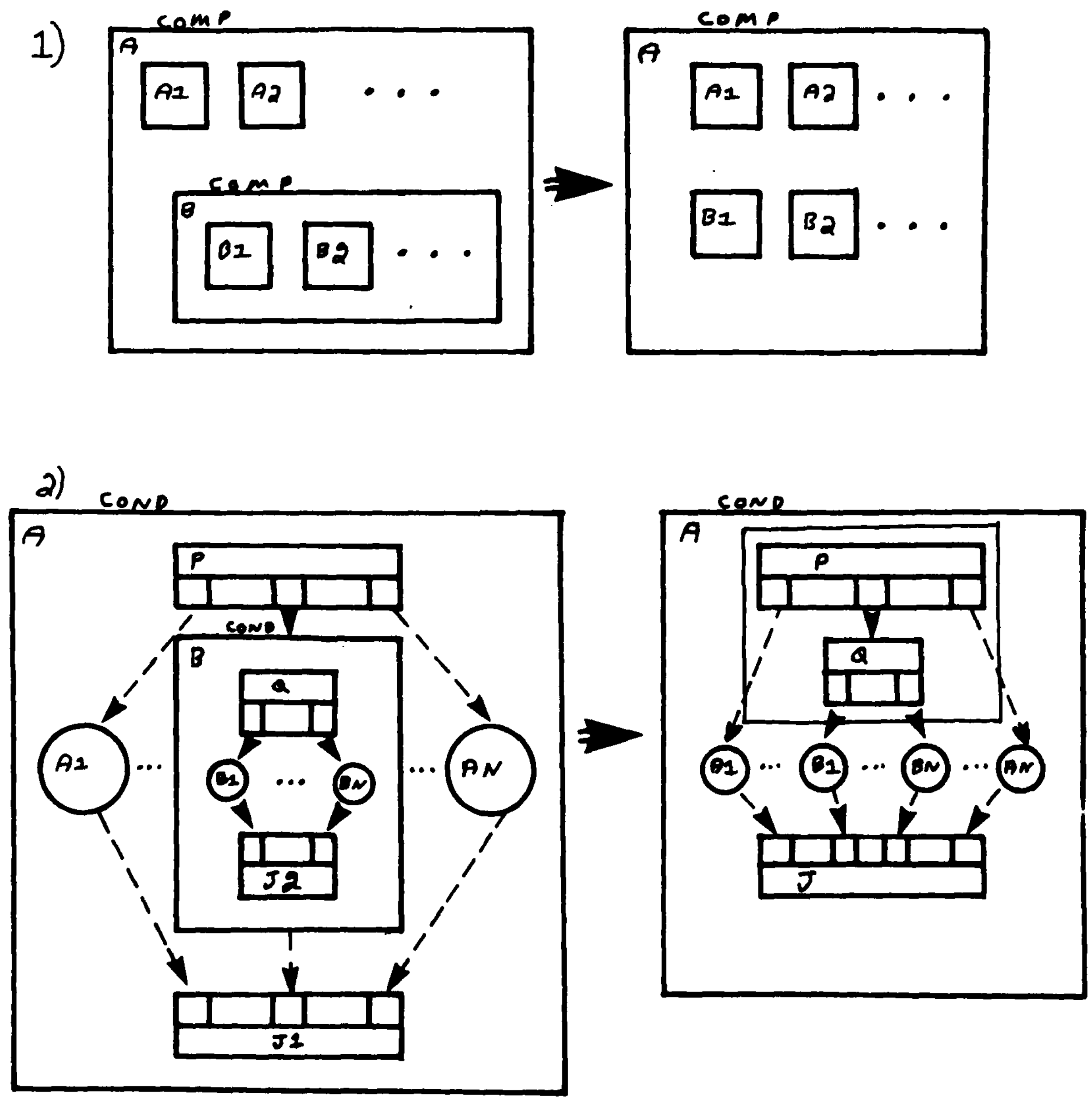


Figure VI-8: Transitive simplifications.

In order to simplify the resulting segmentation, the two transitive simplifications shown in Figure VI-8 are performed. The first one states that if a composition has a subsegment which is also a composition, then the inner composition can be ungrouped producing one large composition. This

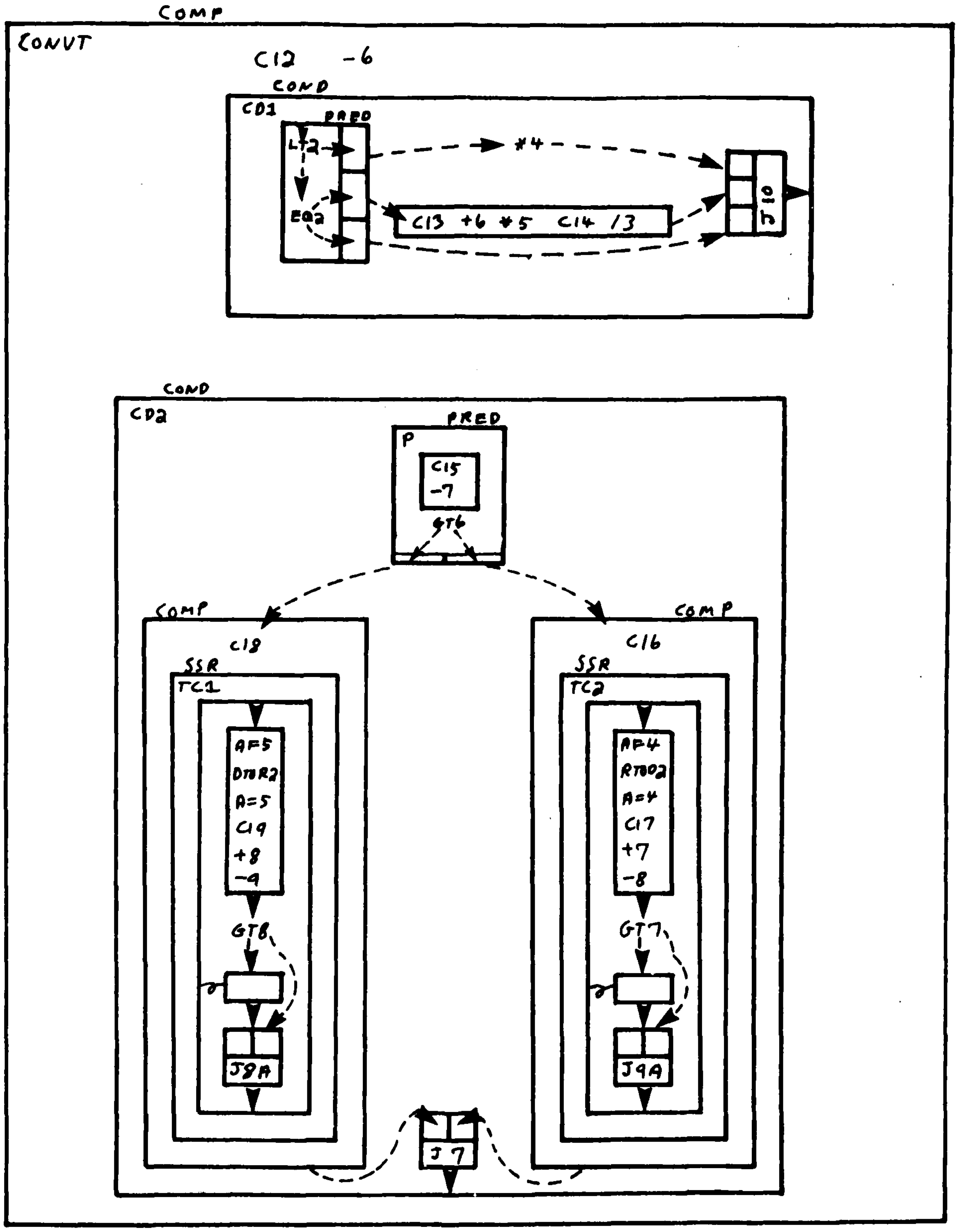


Figure VI-9: Simplified initial segmentation for CONVT.

simplification applies to the example to ungroup the two top level compositions making C12, -6, and the upper conditional CD1 direct subsegments of CONVT, as shown in Figure VI-9.

In a similar vein, the second simplification says that if a conditional has an action which is itself a conditional, then this inner conditional can be ungrouped. The lower split is combined with the upper split in a predicate to produce one big conditional. This applies in the example to coalesce the conditionals involving LT2 and EQ2 as shown in the figure.

A desirable property of an analysis system like the one described here is that the result should depend only on the starting point (here the surface plan), and not on the order in which the steps of recognition are performed. Without the simplifying transformations described above the analyzer would not have this property. For example, notice that in Figure VI-7, there is a composition which groups the composition of C12 and -6 with CD1. If the transformations had been applied in a somewhat different order so that the lower conditional CD2 was grouped up before CD1, then CONVT would have been recognized as a composition of the composition of C12 and -6, and the two conditionals. The simplifying transformations make up for this variability by transforming either of these plans to the same canonical form.

The patterns in Figure VI-10 are used when the simple patterns are not applicable. These patterns are needed in order to analyze a more complex program such as RK1. The first pattern in part A of the figure builds up predicates out of open coded logical expressions such as:

```
IF X>0 THEN GOTO DONE;
IF Y>0 THEN GOTO DONE;
```

or

```
IF X>0 THEN
       IF Y>0 THEN GOTO DONE;
```

Recognizing the pattern would lead to the splits in these examples being grouped together in such a way that their basic similarity to

```
IF X>0 OR Y>0 THEN GOTO DONE;
```

and

```
IF X>0 AND Y>0 THEN GOTO DONE;
```

respectively is made clear.

The second pattern performs the transformation described in Section IV.1.4 in order to swap two splits which are out of order, in that they are blocking further analysis. The two splits can only be swapped if the conditions of case 2 of P2 are implied by the conditions of case 2 of P1. In addition, it must not be the case that P2 is satisfying any pre-conditions of P1. This is the one place where this analysis step uses any knowledge of what a segment is computing. An example of this is the following program:

```
IF X≥0 THEN DO; IF X=0 THEN GOTO DONE;
                 X = FOO(X); END;
       ELSE X = BAR(X);
```

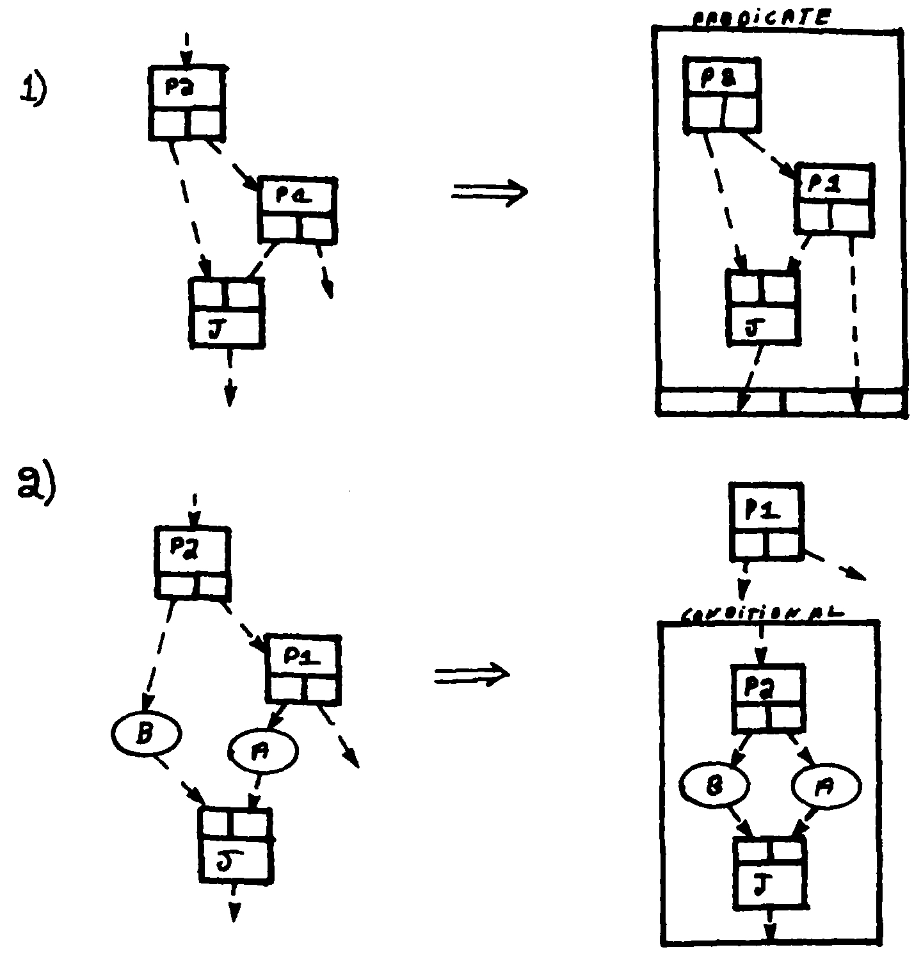


Figure VI-10: The more complex patterns recognized (Part A).

This is transformed to the program:

```
IF X=0 THEN GOTO DONE;
IF X≥0 THEN X = FOO(X);
       ELSE X = BAR(X);
```

which is equivalent. The only examples of this seen so far come form the macro expansion of arithmetic IFs in FORTRAN. The macro expander operates on a completely local basis, and as a result, it is as likely to produce the first program above as the second. The pattern in part B of Figure VI-10 recognizes multi-exit loops.

An important thing to notice about the analysis process described here is that it has no provision for back up. It proceeds without ever taking any wrong turns. This straightforward approach is possible because the system is only looking for four different PBMs, and these PBMs are very different from each other. The fact that the analysis is able to proceed without any back tracking speeds the recognition process.

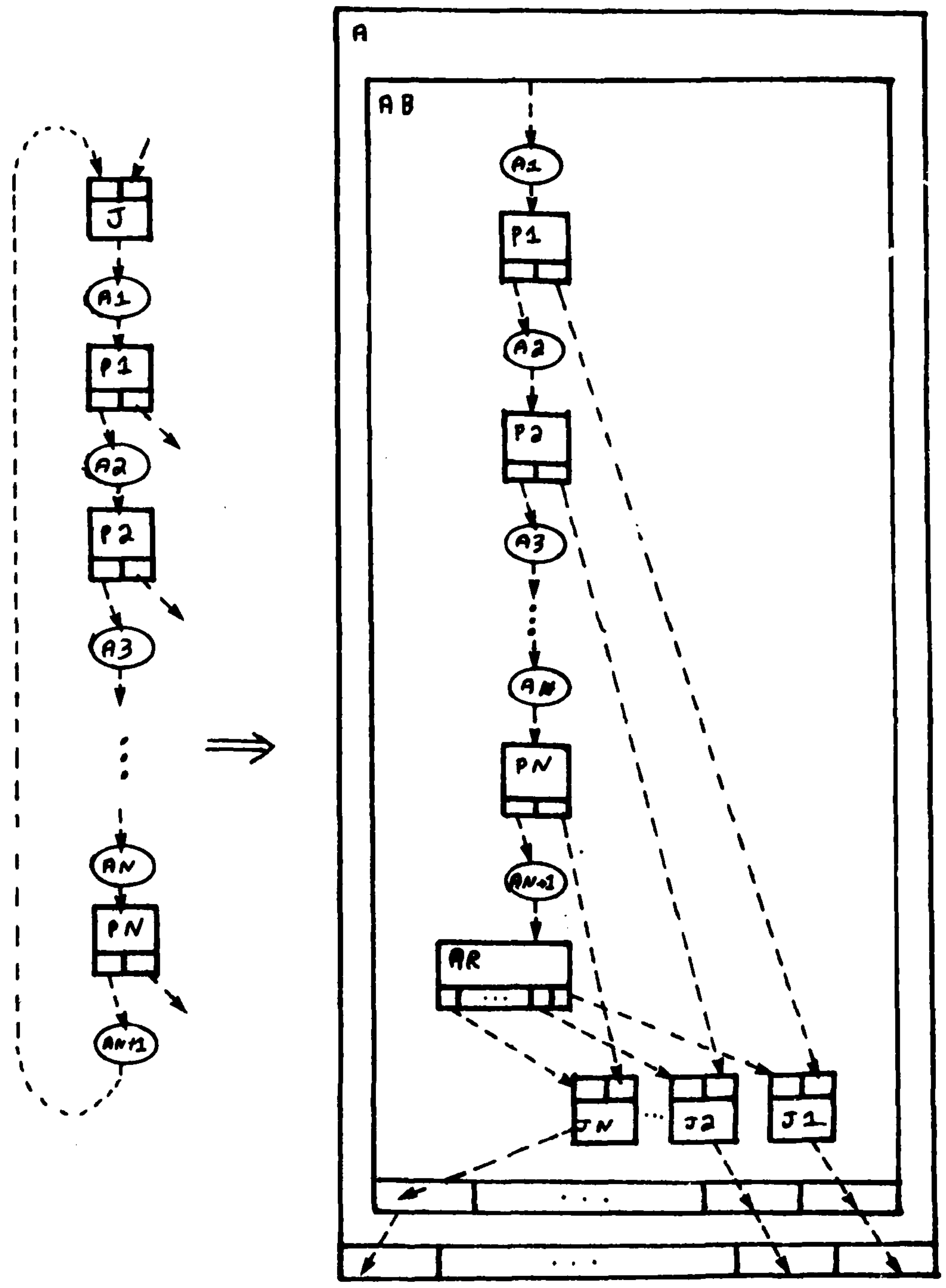


Figure VI-10: The more complex patterns recognized (Part B).

VI.2.2 Step 2: Locating Initializations

The recognition described in the last section is based primarily on control flow. The recognition described here is based primarily on data flow. It is controlled by a single general principle. If all of the data flow leaving a segment (A) goes directly to another segment (B), then A can be moved inside B. The motivation behind this is that since the only place the results of A are used is inside B, A can probably be more easily understood inside B in the environment where its results are used.

If B is a composition then A can simply be moved inside and made an action of B. If B is a predicate, conditional, or single self recursion, then A is moved into the initialization for B, or becomes the initialization for B if B had no previous initialization. If B is a terminal segment then A cannot be put inside B, because B has no inside. A can, however, be grouped together with B into a new segment. If B is a straight-line segment, then the two can be grouped together as a composition. If B is a split, then the two can be grouped together as a predicate with A as the initialization. These last two processes have to be limited so that they will not get into an infinite loop. The problem is that once A and B are grouped together, A still has data flow only to B. This could again trigger grouping A and B together into a new segment. This is prevented by requiring that A and B can be grouped together only if the PBM of the segment to be created is different from the PBM of the segment they are currently in.

There are three situations where the principle above cannot be applied. First, if B is a join then A and B cannot be grouped together, because non-terminal joins are not allowed in a plan. Second, if B is the body of a single self recursion, then A cannot be moved into B. The problem is that, due to the recursive link between the body and the recurrence in a single self recursion, the body acts like a template which describes the computation which is performed on each iteration of the single self recursion. If A was put into the body B, then this would imply that A would be executed on every iteration, while in its initial position outside the body, A is only executed once. In addition, the values of A may only be used on the first iteration, not in every iteration. Third, if B is a recurrence of a single self recursion then A cannot be moved into B. If A is in a position to be moved into a recurrence B then A must already be in the body which is recursively linked to B. As a result of this, A is already in the recurrence which must have the same plan as the body.

The plan in Figure VI-11 shows the plan for the program CONVT which results from moving segments around based on the principle above. The changes which have been made can be seen by comparing this figure with Figure VI-9. In the earlier figure, -6 has data flow only into CD1. As a result, it is moved into CD1 becoming the initialization. Once this is done, it becomes apparent that -6 has data flow only into the predicate which is the split of CD1. As a result of this, it is moved inside the predicate becoming the initialization of the predicate. C12, which has data flow only to -6, follows a similar route, moving into the conditional, and then into the initialization of the predicate. CD1 as a whole has data flow only into CD2, so it becomes the initialization of this conditional. This leaves only one segment, CD2, at the top level of CONVT. The plan is simplified by ungrouping the top level composition, and making conditional the top level PBM in the plan for CONVT.

In the single self recursion on the left (TC1), C18 moves in to become the initialization. Also, -9 (in the body of TC1) has data flow only into the test GT8. As a result, it is moved up out of the composition it is in, and then grouped into a predicate with GT8. The other single self recursion (TC2) undergoes analogous transformations

The transformation process is implemented by simply searching the plan for a segment which has

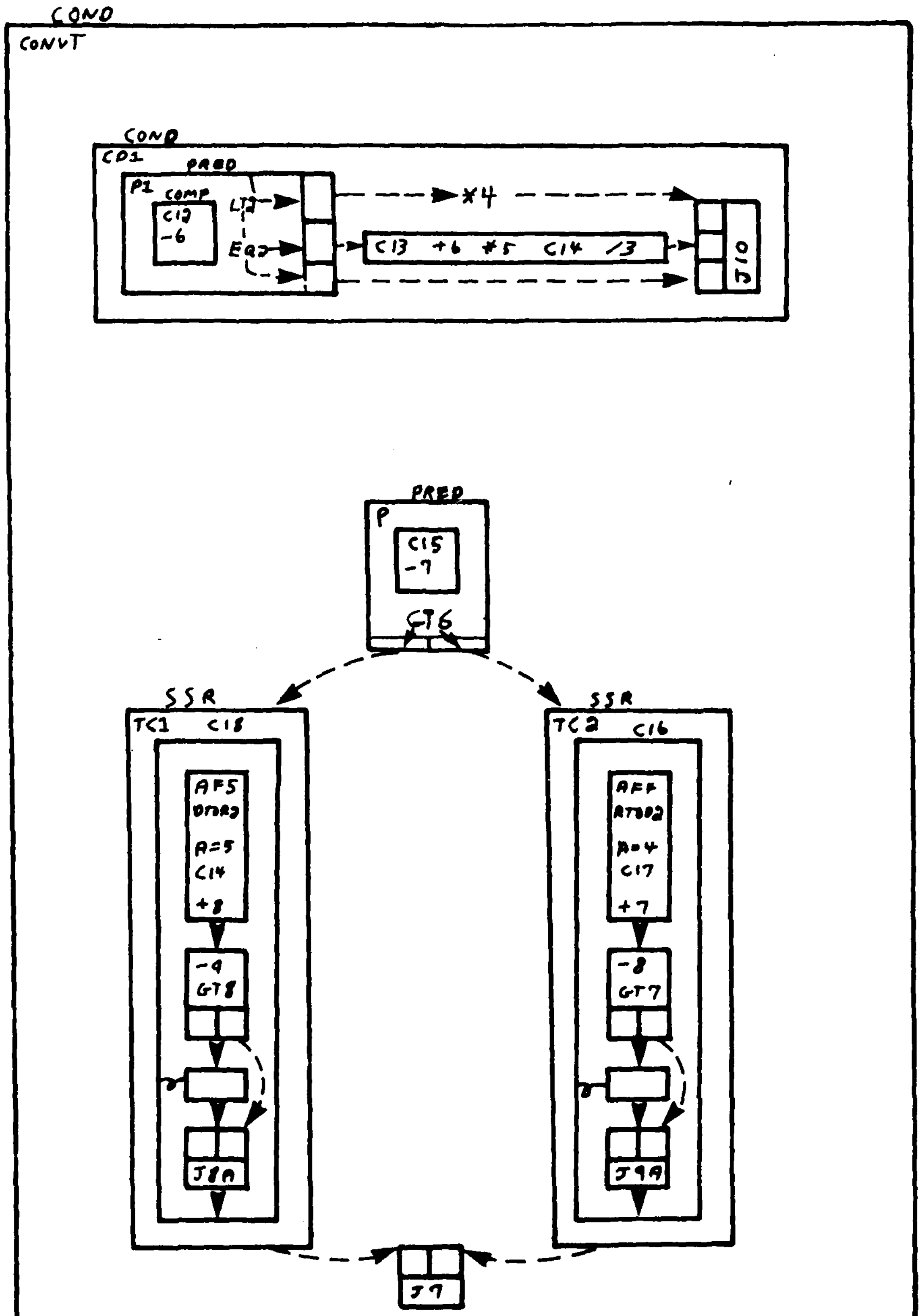


Figure VI-11: Plan for CONVT after initializations have been located.

all of its data flow going to some other segment. When such a segment is found, it is moved, and the search begins again. When a segment is moved, the plan may be changed in such a way that another segment which could not previously be moved can then be moved. (In the example, C12 cannot be moved down into the split of CD1 until after -6 is.) As a result, the search has to pass repeatedly over large areas of the plan. This causes this stage of the analysis to be rather slow. It can often take longer than the first stage.

There are two basic reasons for performing the transformation described in this section. The first reason is that the plan which is produced is "better" in that it seems more natural. One facet of this is that the resulting plan is more uniform. In the plan in Figure VI-9, notice that C15, and -7 are grouped into a predicate with GT6 while C12 and -6 are not grouped with LT2 and EQ2. This difference does not have any philosophical basis. It is just an artifact of the particular patterns which were used to analyze the program. The process of locating initializations described above, moves them all uniformly as close as possible to the place where their outputs are used.

The second reason for doing this transformation is to provide canonicalization of plans. Both the plan in Figure VI-9 and the plan in Figure VI-11 are valid plans for CONVT, as are all of the intermediate plans alluded to in the description of the transformation process, and a host of other plans. The transformation described here singles out one of these plans as the preferred plan for CONVT. Among other things, this canonicalization makes it easier to identify stereotyped parts of the plan.

VI.2.3 Step 3: Analyzing Compositions and Single Self Recursions

The first thing this final step of analysis does, is to look at each composition in the plan and check whether it can be further analyzed as discussed in Section IV.1.1. The data flow is checked to determine whether the subsegments can be partitioned into subsets combined by the PBM conjunction. The analyzer does not try to differentiate between compositions and expressions, because the intermediate segments in the plan being developed do not have any pre-conditions or post-conditions. This distinction is not made until complete behavioral descriptions for these segments are derived as discussed in Chapter VII.

Referring to Figure VI-11, the composition of AF5, DTOR2, A=5, C19, and +8, in the single self recursion TC1, can be partitioned into two groups. One group consists of AF5, DTOR2, and A=5. The other consists of C19 and +8. These two groups are then combined as a conjunction. The composition in the body of the other single self recursion TC2 is analyzed similarly.

The third step then analyzes each single self recursion in terms of the PBMs temporal composition, augmentation, filter, and termination. The analysis is done exactly as described in Section IV.2.1.5.3. This leads to the final plan shown in Figure VI-12. The single self recursion TC1 is analyzed as consisting of three parts: an augmentation which enumerates integers starting with 1, a termination which truncates this sequence at NM, and another augmentation which selects the indicated elements in the array D, converts them to a real numbers, and then puts them into the corresponding positions in array S.

At the present time, the correctness of the analyzer has been shown only through experiment. There are several properties of the analyzer which should be verified. For example, it is desirable that the plan which results from an analysis of a program should not be a function of the order in which the analyzer performs transformations. In addition, the analyzer should be able to successfully analyze any surface plan which is constructed by taking a plan, built up by means of

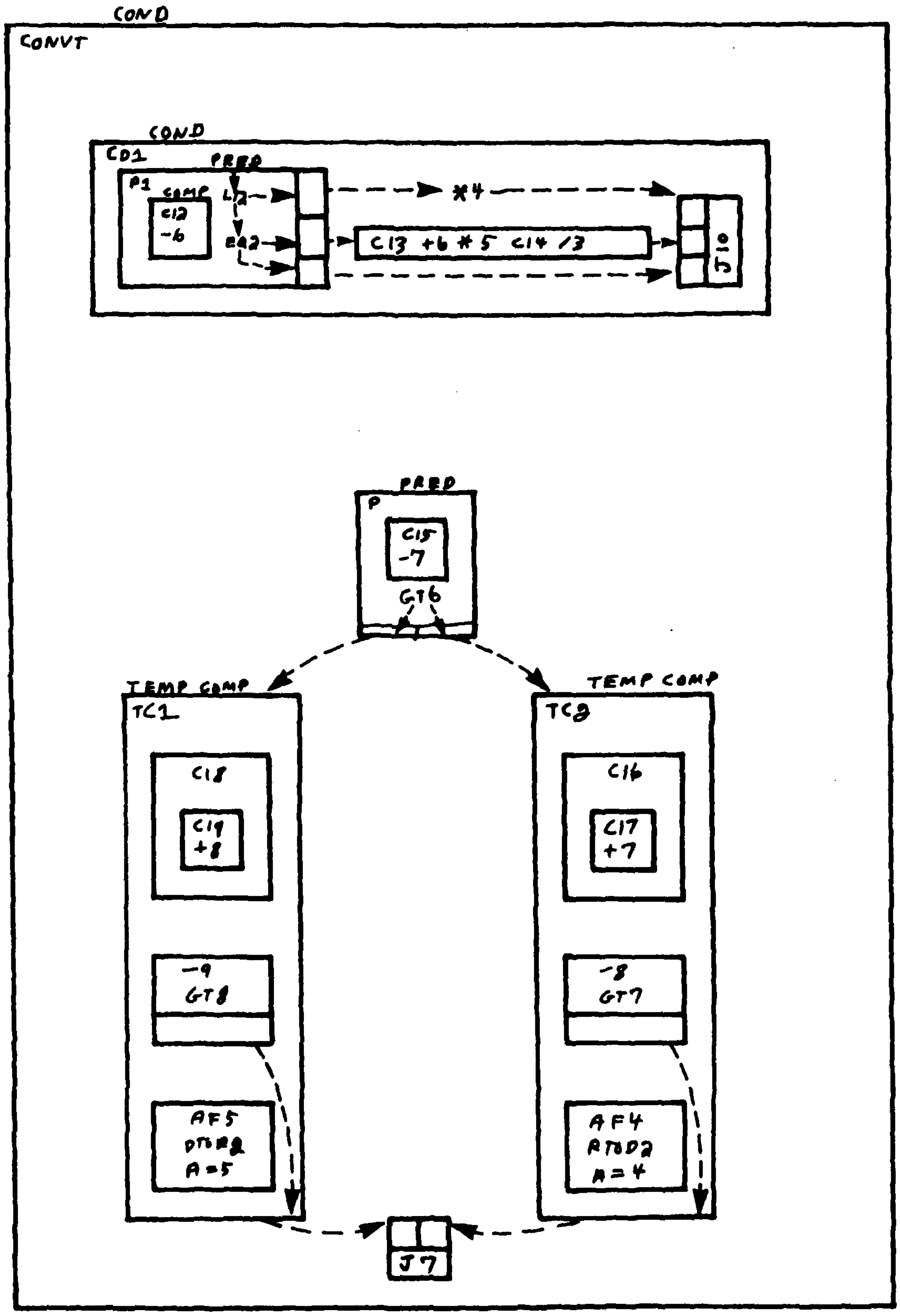


Figure VI-12: The final plan for CONVT.

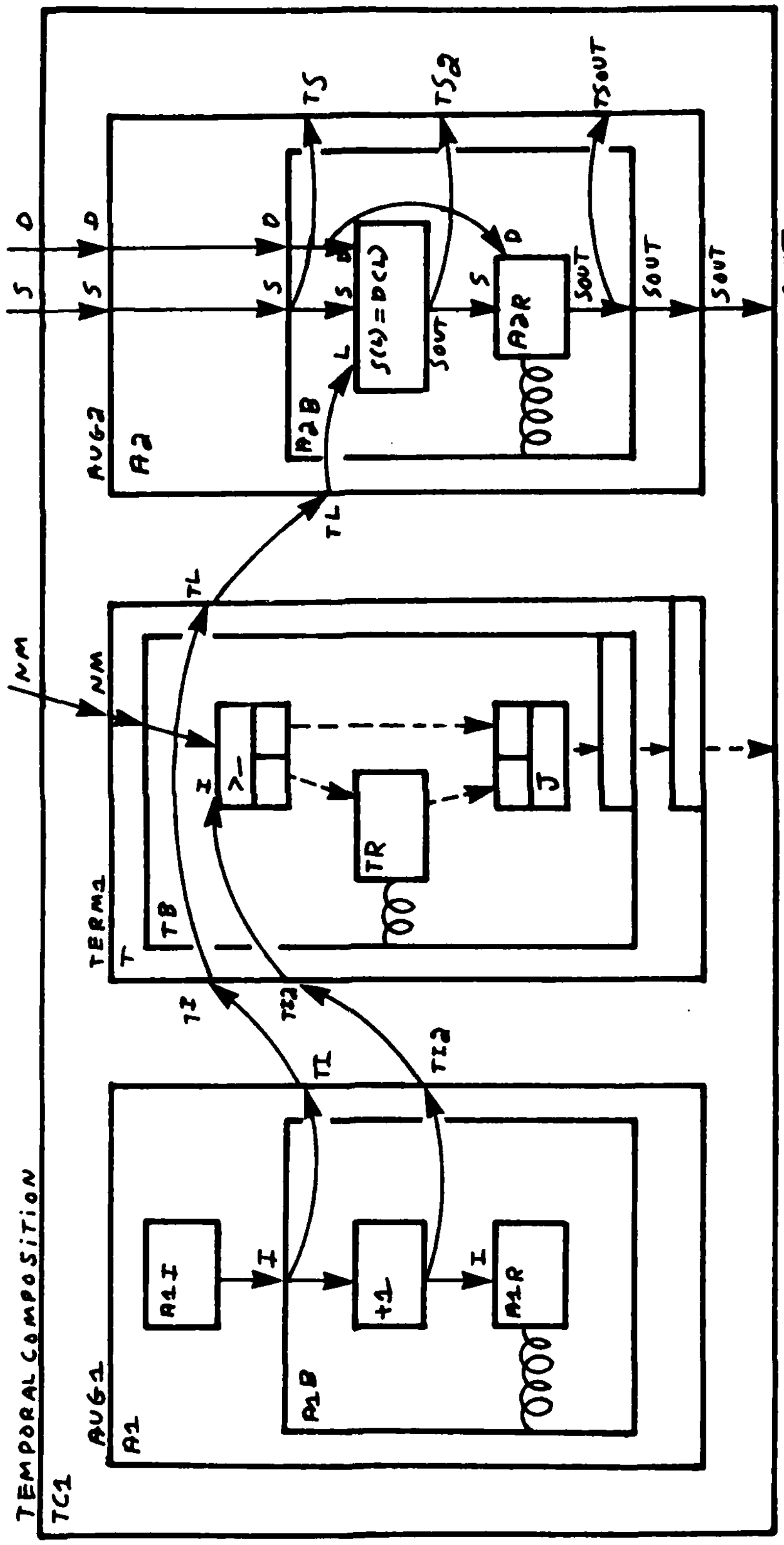
PBMs, and discarding the intermediate segmentation. It should be noted that the plan which results from the analysis will generally not have the same structure as the original plan since the analyzer produces a canonical result. It would also be desirable to have some characterization of what the analyzer can analyze besides the statement that it can analyze any surface plan derived by ungrouping a plan built up by means of PBMs.

VII. Specification Generation

This chapter presents a system, implemented by the author, which automatically generates the specifications for a program. It takes as its input a segmented PBM plan derived by the automatic analysis system described in the last chapter. This plan has a complete behavioral description for each terminal segment, but does not have any pre-conditions or post-conditions for any of the non-terminal segments. Behavioral descriptions for these segments are generated bottom up by using the methods appropriate to each PBM. These individual methods are described in detail in the subsections of Chapter IV. This chapter discusses the specifications which are automatically generated for the program CONVT, and speculates on how useful the specification generation process is in general.

Figure VI-12 shows the PBM plan for CONVT. At the lowest level, there are a number of compositions which all turn out to be simple expressions. Generating their behavioral descriptions is easy. The generation of the behavioral descriptions for the predicates P and P1 and the conditional CD1 is also straightforward. The generation of the behavioral descriptions for the two temporal compositions TC1 and TC2 is more interesting. Figure VII-1 shows the behavioral description which is derived for TC1. The behavioral description for TC2 is closely analogous.

```
                 DO 25 L=1, NM
              25 S(L) = D(L)

          Segment TC1 (the whole loop above)
           inputs: NM, S, D
   pre-conditions: INTEGER(NM), REAL(S), VECTOR(S), DOUBLE(D), VECTOR(D),
                   1≤NM, NM≤SIZE(S), NM≤SIZE(D)
          outputs: SOUT
  post-conditions: REAL(SOUT), VECTOR(SOUT),
                   SOUT=MULTI-ARRAY-ASSIGN_{j=1,NM}(S,j,DTOR(D(j)))
   justifications: (A2.po1)→TC1.po1, (A2.po2)→TC1.po2,
                   (A2.po3, A1.po2, A1.po4, T.po1, T.po2, T1.po2)→TC1.po3
              pbm: temporal composition
            roles: augmentation1 A1, termination1 T, augmentation2 A2
       role dflow: TC1.NM→T.NM, TC1.S→A2.S, TC1.D→A2.D, A2.SOUT→TC1.SOUT
       role cflow: T1→TC
   temporal dflow: A1.TI→T.TI, A1.TI2→T.TI2, T.TJ→A2.TL

          Segment A1
   pre-conditions: last-state(TI)=last-state(TI2)
 temporal outputs: TI=(A1B.I at A1Bi), TI2=(+1.J at +1o)
  post-conditions: ∀i∈TI(INTEGER(TI_i)), ∀i∈TI(TI_i=i),
                   ∀i∈TI2(INTEGER(TI2_i)), ∀i∈TI2(TI2_i=i+1)
   justifications: (T)→A1.pr1
```



Temporal Flow diagram for Figure VII-1.

```
             Segment T
              inputs: NM
     temporal inputs: TI=(at >i), TI2=(>.I at >i)
      pre-conditions: ∀i∈TI2(INTEGER(TI2_i)), INTEGER(NM)
    temporal outputs: TJ=(TI_i at >i)
     post-conditions: ∀i∈TJ(TJ_i=TI_i), last-state(TJ)=last-state(TI2)=last-state(TI)
      justifications: (A1.po3)→T.pr1, (TC1.pr1)→T.pr2
         CASE1
          conditions: ∃i∈TI2(TI2_i>NM)
     post-conditions: terminates(T), last-state(TI2)=MIN({i∈TI2| TI2_i>NM})
         CASE2
          conditions: ¬∃i∈TI2(TI2_i>NM)
     post-conditions: ¬terminates(T)

             Segment A2
              inputs: S, D
     temporal inputs: TL=(A2AF.L at A2AFi)
      pre-conditions: REAL(S), VECTOR(S), DOUBLE(D), VECTOR(D),
                      ∀i∈TL(INTEGER(TL_i)), ∀i∈TL(TL_i≤SIZE(S)), ∀i∈TL(TL_i≤SIZE(D)),
                      last-state(TSOUT)=last-state(TS),
                      TSOUT_last-state(TSOUT)=TS2_last-state(TS2)
             outputs: SOUT
    temporal outputs: TS=(A2B.S at A2Bi), TS2=(A2AF.SOUT at A2AFo),
                      TSOUT=(A2B.SOUT at A2Bo)
     post-conditions: REAL(SOUT), VECTOR(SOUT),
                      SOUT=MULTI-ARRAY-ASSIGN_j=1,last-state(TL)(S,TL_j,DTOR(D(TL_j)))
      justifications: (TC1.pr2)→A2.pr1, (TC1.pr3)→A2.pr2, (TC1.pr4)→A2.pr3,
                      (TC1.pr5)→A2.pr4, (T.po1, A1.po1)→A2.pr5,
                      (TC1.pr7, T.po1, T.po2, T1.po2, A1.po2, A1.po4)→A2.pr6,
                      (TC1.pr8, T.po1, T.po2, T1.po2, A1.po2, A1.po4)→A2.pr7,
                      (T)→A2.pr8, (T)→A2.pr9
```

Figure VII-1: The plan for the loop TC1 in CONVT.

TC1 is composed of three parts: an augmentation (A1) which enumerates integers, a termination (T) which truncates this sequence at NM and a second augmentation (A2) which does an array assignment. A1 and T are very simple to understand. A2 is more complex. It is a composition augmentation since it has data flow to itself through the array S. The basic statement of what happens on an iteration of A2 is "$TS2_i$=ARRAY-ASSIGN(TS_i, TL_i, DTOR(D(TL_i)))". This says that the resulting array $TS2_i$ is gotten from the initial array TS_i by changing the value of the TL_i component to DTOR(D(TL_i)), and leaving the other elements unchanged. It is also clear that "TS_1=S", and "TS_i=$TS2_{i-1}$". Together these form a recurrence relation which does not in general have a simple solution.

The operation MULTI-ARRAY-ASSIGN is specifically introduced to express the solutions to recurrence relations involving array assignment. Suppose "TA_1=A" and "TA_i=ARRAY-ASSIGN(TA_{i-1}, TJ_{i-1}, TK_{i-1})". Then "TA_i=MULTI-ARRAY-ASSIGN$_{j=1,i-1}$(A, TJ_j, TK_j)". This implies that TA_i is the same size as A. Further, "TA_i(K)=A(K)" if there is no TJ_j equal to K. Otherwise, "TA_i(K)=TK_j" where j is the largest subscript such that "TJ_j=K". It should be noted that MULTI-ARRAY-ASSIGN is a rather intractable operation, and it is difficult to reason about it except in situations where there are no duplicates in the sequence of TJs, as is the case in CONVT.

The operation MULT-ARRAY-ASSIGN can be used to summarize the actions of A2 as "SOUT=MULTI-ARRAY-ASSIGN$_{j=1,\text{last-state}(TL)}$(S, TL_j, DTOR(D(TL_j)))". The behavioral description of TC1 can then be derived by composing together the behavioral descriptions of A1, T and A2 (see Section IV.2.1.5.2). Substitution yields the post-condition that TC1 computes

"SOUT=MULTI-ARRAY-ASSIGN$_{j=1,NM}$(S, j, DTOR(D(j)))". The pre-condition "1≤NM" comes from the fact that in the semantics of FORTRAN, the test in a DO loop is at the bottom of the loop and therefore TC1 will not operate correctly if NM equals zero. It should be noted that justification links are derived as a natural part of the behavioral description generation process.

```
      SUBROUTINE CONVT (N,M,MODE,S,D,MS)
      DIMENSION S(1),D(1)
      DOUBLE PRECISION D
      IF (MS-1) 2, 4, 6
    2 NM=N*M
      GO TO 8
    4 NM=((N+1)*N)/2
      GO TO 8
    6 NM=N
    8 IF (MODE-1) 10, 10, 20
   10 DO 15 L=1,NM
   15 D(L)=S(L)
      GO TO 30
   20 DO 25 L=1,NM
   25 S(L)=D(L)
   30 RETURN
      END
```

The role diagram is the same as the one in Figure VI-12.

```
          Segment CONVT (the whole program above)
           inputs: S, D, N, M, MODE, MS
   pre-conditions: REAL(S), VECTOR(S), DOUBLE(D), VECTOR(D),
                   INTEGER(N), INTEGER(M), INTEGER(MS), INTEGER(MODE),
                   1≤(IF MS<1 THEN N*M IF MS=1 THEN ((N+1)*N)/2 IF MS>1 THEN N)
                   (IF MS<1 THEN N*M IF MS=1 THEN ((N+1)*N)/2 IF MS>1 THEN N)≤SIZE(S)
                   (IF MS<1 THEN N*M IF MS=1 THEN ((N+1)*N)/2 IF MS>1 THEN N)≤SIZE(D)
          outputs: SOUT, DOUT
  post-conditions: REAL(SOUT), VECTOR(SOUT), DOUBLE(DOUT), VECTOR(DOUT),
                   SOUT=(IF MODE≤1 THEN S
                         IF MODE>1 THEN MULTI-ARRAY-ASSIGNj=1,NM (S,j,DTOR(D(j)))
                     where NM=(IF MS<1 THEN N*M IF MS=1 THEN ((N+1)*N)/2 IF MS>1 THEN N)),
                   DOUT=(IF MODE≤1 THEN MULTI-ARRAY-ASSIGNj=1,NM (D,j,RTOD(S(j)))
                     where NM=(IF MS<1 THEN N*M IF MS=1 THEN ((N+1)*N)/2 IF MS>1 THEN N)
                         IF MODE>1 THEN D)
   justifications: (TC1.po1, J71.po1, CONVT.pr1, J72.po1)→CONVT.po1,
                   (TC1.po2, J71.po1, CONVT.pr2, J72.po1)→CONVT.po2,
                   (TC2.po1, J72.po2, CONVT.pr3, J71.po2)→CONVT.po3,
                   (TC2.po2, J72.po2, CONVT.pr4, J71.po2)→CONVT.po4,
                   (CD1.po2, P1.c1, TC1.po3, J71.po1, P2.c1, J72.po1)→CONVT.po5,
                   (CD1.po2, P2.c1, TC2.po3, J72.po2, P1.c1, J71.po2)→CONVT.po6
              pbm: conditional
            roles: initialization CD1, split P, action1 TC1, action2 TC2, join J7
       role dflow: CONVT.S→TC1.S, CONVT.S→TC2.S, CONVT.S→J72.S2, CONVT.D→TC1.D,
                   CONVT.D→TC2.D, CONVT.D→J71.D1, CONVT.N→CD1.N, CONVT.M→CD1.M,
                   CONVT.MODE→P.MODE, CONVT.MS→CD1.MS, CD1.NM→TC1.NM,
                   CD1.NM→TC2.NM, TC1.SOUT→J71.S1, TC2.DOUT→J72.D2
       role cflow: P1→TC1, P2→TC2, TC1→J71, TC2→J72, J7→CONVT
```

```
          Segment CD1
           inputs: N, M, MS
   pre-conditions: INTEGER(N), INTEGER(M), INTEGER(MS)
          outputs: NM
  post-conditions: INTEGER(NM),
                   NM=(IF MS<1 THEN N*M IF MS=1 THEN ((N+1)*N)/2 IF MS>1 THEN N)
   justifications: (CONVT.pr5)→CD1.pr1, (CONVT.pr6)→CD1.pr2, (CONVT.pr7)→CD1.pr3

          Segment P
           inputs: MODE
   pre-conditions: INTEGER(MODE)
      CASE1
       conditions: MODE>1
      CASE2
       conditions: MODE≤1
   justifications: (CONVT.pr8)→P.pr1

          Segment TC1
           inputs: NM, S, D
   pre-conditions: INTEGER(NM), REAL(S), VECTOR(S), DOUBLE(D), VECTOR(D),
                   1≤NM, NM≤SIZE(S), NM≤SIZE(D)
          outputs: SOUT
  post-conditions: REAL(SOUT), VECTOR(SOUT),
                   SOUT=MULTI-ARRAY-ASSIGN_{j=1,NM}(S,j,DTOR(D(j)))
   justifications: (CD1.po1)→TC1.pr1, (CONVT.pr1)→TC1.pr2, (CONVT.pr2)→TC1.pr3,
                   (CONVT.pr3)→TC1.pr4, (CONVT.pr4)→TC1.pr5,
                   (CD1.po2, CONVT.pr9)→TC1.pr6, (CD1.po2, CONVT.pr10)→TC1.pr7,
                   (CD1.po2, CONVT.pr11)→TC1.pr8

          Segment TC2
           inputs: NM, S, D
   pre-conditions: INTEGER(NM), REAL(S), VECTOR(S), DOUBLE(D), VECTOR(D),
                   1≤NM, NM≤SIZE(S), NM≤SIZE(D)
          outputs: DOUT
  post-conditions: REAL(DOUT), VECTOR(DOUT),
                   DOUT=MULTI-ARRAY-ASSIGN_{j=1,NM}(D,j,RTOD(S(j)))
   justifications: (CD1.po1)→TC2.pr1, (CONVT.pr1)→TC2.pr2, (CONVT.pr2)→TC2.pr3,
                   (CONVT.pr3)→TC2.pr4, (CONVT.pr4)→TC2.pr5,
                   (CD1.po2, CONVT.pr9)→TC2.pr6, (CD1.po2, CONVT.pr10)→TC2.pr7,
                   (CD1.po2, CONVT.pr11)→TC2.pr8

          Segment J7
          outputs: SOUT, DOUT
      CASE1
           inputs: S1, D1
  post-conditions: SOUT=S1, DOUT=D1
      CASE2
           inputs: S2, D2
  post-conditions: SOUT=S2, DOUT=D2
```

Figure VII-2: The plan for the top level of CONVT.

Once behavioral descriptions for TC1 and TC2 have been derived, a behavioral description and justification links for CONVT as a whole can be derived as shown in Figure VII-2. This derivation is straightforward, and follows the pattern described in Section IV.1.3 which describes conditionals.

The behavioral description for CONVT which is produced is accurate, and clearly follows from the logical structure of the program. However, it is not very satisfying. The problem is that it is very unwieldy. There are basically two reasons for this. First, the behavioral description derived does not use any of the high level concepts associated with the program. The header comment for the program (see Figure VII-3) shows what some of these concepts are. For example, what CD1 is doing can be summarized by saying that it is computing the size of an N by M matrix with storage mode MS. The FORTRAN subroutines in the SSP store matrices in vectors in three different ways based on whether they are triangular (MS=1), diagonal (MS=2) or general (MS=0).

The second reason that the derived specifications are unwieldy is that like most programs, CONVT is not expected to do as much as it is actually capable of doing. For example, as the header comment shows, the two vectors S and D are both expected to be N by M matrices with storage mode MS. Part of the unwieldiness of the behavioral description comes from the fact that it describes what CONVT is actually capable of doing. For example, it describes what CONVT would do given two matrices of different sizes greater than the value calculated in NM.

header comment for CONVT

```
C       PURPOSE
C          CONVERT NUMBERS FROM SINGLE PRECISION TO DOUBLE PRECISION
C          OR FROM DOUBLE PRECISION TO SINGLE PRECISION.
C       DESCRIPTION OF PARAMETERS
C          N    - NUMBER OF ROWS IN MATRICES S AND D.
C          M    - NUMBER OF COLUMNS IN MATRICES S AND D.
C          MODE - CODE INDICATING TYPE OF CONVERSION
C                    1 - FROM SINGLE PRECISION TO DOUBLE PRECISION
C                    2 - FROM DOUBLE PRECISION TO SINGLE PRECISION
C          S    - IF MODE=1, THIS MATRIX CONTAINS SINGLE PRECISION
C                 NUMBERS AS INPUT.  IF MODE=2, IT CONTAINS SINGLE
C                 PRECISION NUMBERS AS OUTPUT.  THE SIZE OF MATRIX S
C                 IS N BY M.
C          D    - IF MODE=1, THIS MATRIX CONTAINS DOUBLE PRECISION
C                 NUMBERS AS OUTPUT.  IF MODE=2, IT CONTAINS DOUBLE
C                 PRECISION NUMBERS AS INPUT.  THE SIZE OF MATRIX D IS
C                 N BY M.
C          MS   - ONE DIGIT NUMBER FOR STORAGE MODE OF MATRIX
C                    0 - GENERAL
C                    1 - SYMMETRIC
C                    2 - DIAGONAL
C       METHOD
C          ACCORDING TO THE TYPE OF CONVERSION INDICATED IN MODE, THIS
C          SUBROUTINE COPIES NUMBERS FROM MATRIX S TO MATRIX D OR FROM
C          MATRIX D TO MATRIX S.
```

global knowledge:

```
SIZE(A)=(IF STORAGE-MODE(A)=0 THEN NUMBER-ROWS(A)*NUMBER-COLUMNS(B)
         IF STORAGE-MODE(A)=1 THEN ((NUMBER-ROWS(A)+1)*NUMBER-ROWS(A))/2
         IF STORAGE-MODE(A)=2 THEN NUMBER-ROWS(A))
B=COMPONENTWISE(F,A) ≡
```

$$B=\text{MULTI-ARRAY-ASSIGN}_{i=1,\text{SIZE}(A)}(B,i,F(A(i))) \wedge \text{SIZE}(B)=\text{SIZE}(A)$$

```
         Segment CONVT
          inputs: S, D, N, M, MODE, MS
  pre-conditions: REAL(S), VECTOR(S), DOUBLE(D), VECTOR(D),
                  INTEGER(MS), INTEGER(N), INTEGER(M),
                  INTEGER(MODE), MODE∈{1,2},
                  MS=STORAGE-MODE(S)=STORAGE-MODE(D),
                  N=NUMBER-ROWS(S)=NUMBER-ROWS(D),
                  M=NUMBER-COLUMNS(S)=NUMBER-COLUMNS(D)
         outputs: SOUT, DOUT
 post-conditions: REAL(SOUT), VECTOR(SOUT), DOUBLE(DOUT), VECTOR(DOUT),
                  MS=STORAGE-MODE(SOUT)=STORAGE-MODE(DOUT),
                  N=NUMBER-ROWS(SOUT)=NUMBER-ROWS(DOUT),
                  M=NUMBER-COLUMNS(SOUT)=NUMBER-COLUMNS(DOUT),
                  SOUT=(IF MODE=1 THEN S IF MODE=2 THEN COMPONENTWISE(DTOR, D)),
                  DOUT=(IF MODE=1 THEN COMPONENTWISE(RTOD, S) IF MODE=2 THEN D)
```

Figure VII-3: A better behavioral description for CONVT.

The behavioral description in Figure VII-3 is an improved behavioral description for CONVT based on the header comment which was written as part of the program. The major differences between this behavioral description and the last are the fact that the pre-conditions are more restrictive, the use of the operation COMPONENTWISE, and the use of the relationship between the size of a matrix and its storage mode and dimensions. These things lead to a much more compact behavioral description. The improved behavioral description could be associated with CONVT by using the PBM new-view. Further it would not be difficult to show that it is an accurate behavioral description given the two pieces of global knowledge indicated. However, it is not clear that there is any way to automatically generate the improved behavioral description based solely on the code for the program CONVT, because it depends on what the programmer had in mind.

Stepping back, consider when the kind of automatic generation of behavioral descriptions described here is liable to be useful. The behavioral descriptions are constructed by pushing information from the behavioral descriptions of the terminal segments up to higher and higher levels. As behavioral descriptions are built up at higher and higher levels, problems due to lack of higher level knowledge, and excessive generality, mount up until the behavioral descriptions become more and more inappropriate. As long as the generation process extends only across three, four (as in CONVT), or even five levels, the behavioral descriptions which result are not too unreasonable. However, going further than this leads to behavioral descriptions which are not liable to be useful. For example, the behavioral description automatically generated for RK1, which has seven levels, is pages long.

The PA is designed to utilize the kind of behavioral description generation described here. However, it also uses comments provided by the programmer so that behavioral descriptions do not have to be generated across more than a few levels. The automatic generation is used essentially to fill in the gaps in what the programmer says, and to link what he says to the program.

VIII. A Mini Programmer's Apprentice

This chapter describes how the PBMs could be used as the primary basis for a mini-PA designed to operate in the restricted domain of mathematical FORTRAN programs such as those in the SSP [49]. In order to see how this can be done, consider how the PA as a whole is designed to operate. The operation of the PA is based on six sources of knowledge as shown in Figure VIII-1. This knowledge is used to construct plans for the program being worked on. These plans are stored in the "design notebook" which is used to store all of the knowledge the PA has about a particular program.

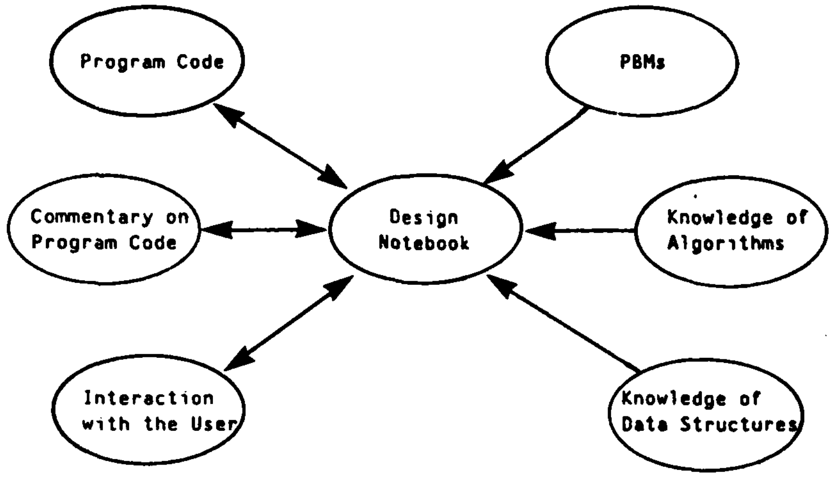

Program Code
PBMs
Commentary on Program Code
Design Notebook
Knowledge of Algorithms
Interaction with the User
Knowledge of Data Structures

Figure VIII-1: Knowledge sources used by the PA.

The circles on the right of the figure represent the three types of knowledge which the PA has about programming in general. It has knowledge about how complex programs are built up out of simpler ones. This knowledge is organized around the PBMs as discussed above. It specifies how segments can be combined together, how a program can be analyzed, and how specifications can be generated for a segment. The PA also has specific knowledge about common algorithms in the form of detailed plans for them. The PA utilizes this knowledge by recognizing instances of these algorithms in the program it is working on. The PA's third type of general knowledge is knowledge of common data structures. This is also utilized by recognizing instances of it in a program. The specific knowledge of data structures and algorithms is combined together into a single component called the "plan library". The knowledge is organized in a way similar to data abstractions so that the relationship between a data structure and the operations which can be performed on it is made clear. Charles Rich [79, 80, 82] is investigating what information should be in the plan library, and how it should be represented.

The circles on the left of the figure represent the three sources of information which the PA has about the particular program it is working on. The first source is the actual code for the program. If the user writes a program, the PA will translate it into a surface plan which is entered into the design notebook along with the code itself. Given a detailed plan for a program, the PA is able to generate code for the program. As a result, code can also be one of the outputs of the PA system. The second source of information about a program is commentary on the program written by the programmer. Most commentary indicates parts of specifications for segments in a plan. However, it can also be used to guide analysis and recognition. Given a detailed plan, the PA can generate commentary as well as code. The third source of information is interaction with the user. The PA can ask the programmer questions. The programmer can also ask the PA questions. Answering questions and making comments is the major output of the PA.

There are four primary active components in the PA: reasoning, analysis, recognition, and interaction with the user. The reasoning component is used by all of the other parts of the system. It is specially designed to work in the domain of plans. A prototype reasoning component has been constructed by Howard Shrobe [79] who is designing an improved version [91, 92]. The analysis component is based on PBMs and operates on surface plans, breaking them apart into pieces which are small enough so that they can be successfully attacked by the other components of the PA. A prototype of this system has been implemented by the author as described in Chapter VI. The recognition component identifies instances of algorithms and data structures from the plan library in a program. Once something has been recognized, all of the knowledge the PA has about it can be added into the plan in the design notebook. The interaction component answers questions and makes comments based on the knowledge in the design notebook. It is discussed more fully below.

The pivotal activity of the PA is the construction of a plan for the program being worked on. This is done by a combination of analysis, recognition, reasoning, and interaction with the user. This chapter concentrates on the mode of interaction between the PA and a programmer which is initiated by the programmer's writing of a program. Once he has done this, the PA's first goal is to construct a plan for the program which has intermediate segmentation, specifications for each segment, and justification links supporting those specifications. It is assumed that the user will provide the major part of the specifications for the program as a whole. The analysis system creates intermediate segmentation. The recognition component identifies segments which correspond to known algorithms. When it proposes a match, the reasoning component is called to verify the match. Once a match has been confirmed, specifications and justification links can be copied from the plan library. Analysis can be used to create specifications by propagating the specifications of subsegments upward as described in Chapter VII. This process also creates justification links for the generated specifications. Consultation with the user is used to determine specifications for segments in situations where recognition is not possible and where the generated specifications are not appropriate. The user can indicate the specifications in comments, or in response to questions posed by the PA. Once a segment has been given specifications, the reasoning system can be used to create justification links.

An important aspect of the process described above, is that several different components can be used to achieve each part of the goal. Further, each component is powerful enough so that there are situations in which it can be used to do all of the work. Though it is excessively tedious, interaction with the user can always be used to construct a complete plan. If the user breaks up his program into sufficiently simple subroutines and gives each one a set of specifications, then the reasoning system can be used to construct justification links for each one without the need for any more segmentation. Similarly, if the user breaks the program up into subroutines corresponding to algorithms known to the PA, recognition can be used to provide a complete plan. Finally, if each subroutine is sufficiently straightforward, analysis and specification generation can be used to create specifications and justification links. The fact that there is so much overlap in the capabilities of the four components provides a firm basis for believing that, together, the four components can construct a complete plan for a program in a wide range of situations.

This chapter focuses on one class of programs, and shows how a simple mini-PA, which has no recognition component and only a very simple reasoning component, could be constructed to work with these programs. The class of programs is restricted to mathematical FORTRAN programs such as those in the SSP. The only data types used in these programs are numbers, vectors, and matrices. As a result, the knowledge base of common data structures can be largely eliminated. The small amount of knowledge which remains can be procedurally embedded as special case knowledge in the rest of the system.

The algorithms used in these programs fall primarily into two categories. First, a small number of very simple algorithms appear very frequently. An example of this is the fact that five basic augmentations accounted for 68% of the augmentations seen in the experiment described in Chapter V. These very common algorithms can be taken out of the plan library and procedurally embedded as special cases in the specification generation process. Most of the rest of the algorithms correspond to mathematical equations and do not occur more than once. Without a knowledge of mathematics, there is nothing more to be said about them except that the program computes them. An example of this is a formula for doing numerical integration, or approximating the hyperbolic SIN. As a result of these simplifications, no algorithms need to be placed in the plan library. The library, along with the entire recognition process, can be eliminated. Analysis and specification generation can be used to construct the complete plan. In situations where the generated specifications are inappropriate, the user must supply specifications.

Due to the fact that the recognition system has been eliminated, the reasoning system is not needed to verify matches. The analysis and specification generation processes require only very simple reasoning. All they needed is a system which can reason about substitution of equals for equals, and the ability to detect that logical expressions are equivalent in situations where their lexical shape is very similar. A more general reasoning system is still needed to verify specifications given by the user. For this task, the mini-PA uses a reasoning system which utilizes numerical testing in order to search for counter examples. This is practical partly due to the fact that the programs in question only use simple data structures.

The mini-PA described here operates in exactly the same manner as the full PA. The only difference is that the recognition component is missing, and that the reasoning component is much less powerful. AS a result, for the most part, the discussion below applies equally well to the mini-PA and the full PA. This chapter is included in this report both to show how the full PA will work, and to highlight the roles the PBMs play in the full PA by showing that in a restricted domain they can be used as the primary component of a PA. The discussion below talks about what the mini-PA could do and how it could do it. The discussion centers around examples based on the SSP program RK1 which was used as an example in Chapter V. Figures VIII-2 and VIII-3 duplicate figures V-3 and V-4 respectively. The discussion of the earlier figures describes how RK1 performs numerical integration. The only parts of the mini-PA which have been implemented to date are the analysis and specification generation systems discussed in chapters VI and VII.

```
C     PURPOSE
C        INTEGRATES A FIRST ORDER DIFFERENTIAL EQUATION
C        DY/DX=FUN(X,Y) UP TO A SPECIFIED FINAL VALUE
C     DESCRIPTION OF PARAMETERS
C        FUN -USER-SUPPLIED FUNCTION SUBPROGRAM WITH
C             ARGUMENTS X,Y WHICH GIVES DY/DX
C        HI  -THE STEP SIZE
C        XI  -INITIAL VALUE OF X
C        YI  -INITIAL VALUE OF Y WHERE YI=F(XI)
C        XF  -FINAL VALUE OF X
C        YF  -FINAL VALUE OF Y
C        ANSX-RESULTANT VALUE OF X
C        ANSY-RESULTANT VALUE OF Y
C             EITHER ANSX=XF OR ANSY=YF DEPENDING
C             ON WHICH IS REACHED FIRST
C        IER -ERROR CODE
C             IER=0 NO ERROR
C             IER=1 STEP SIZE IS ZERO
C     REMARKS
C        IF XI IS GREATER THAN OR EQUAL TO XF, ANSX=XI AND ANSY=YI
C        IF HI IS ZERO, IER=1, ANSX=XI, AND ANSY=0.0
C     METHOD
C        USES FOURTH ORDER RUNGE-KUTTA INTEGRATION
C        PROCESS ON A RECURSIVE BASIS.  PROCESS IS
C        TERMINATED AND FINAL VALUE ADJUSTED WHEN
C        EITHER XF OF YF IS REACHED
C
      SUBROUTINE RK1(FUN,HI,XI,YI,XF,YF,ANSX,ANSY,IER)
C     IF XF IS LESS THAN OR EQUAL TO XI, RETURN (XI,YI)
      IER=0
      IF (XF-XI) 11,11,12
   11 ANSX=XI
      ANSY=YI
      RETURN
C     TEST INTERVAL VALUE
   12 H=HI
      IF (HI) 16,14,20
   14 IER=1
      ANSX=XI
      ANSY=0.0
      RETURN
   16 H=-HI
C     SET XN=INITIAL X, YN=INITIAL Y
   20 XN=XI
      YN=YI
C     INTEGRATE ONE TIME STEP
      HNEW=H
      JUMP=1
      GOTO 170
   25 XN1=XX
      YN1=YY
C     COMPARE XN1 (=X(N+1)) TO X FINAL AND BRANCH ACCORDINGLY
      IF (XN1-XF) 50,30,40
C     XN1=XF, RETURN (XF,YN1) AS ANSWER
   30 ANSX=XF
      ANSY=YN1
      GOTO 160
C     XN1 GREATER THAN X FINAL, SET NEW STEP SIZE AND
C     INTEGRATE ONE STEP, RETURN RESULTS AS ANSWER
   40 HNEW=XF-XN
      JUMP=2
      GOTO 170
```

```
     45 ANSX=XX
        ANSY=YY
        GOTO 160
C       XN1 LESS THAN X FINAL, CHECK IF (YN,YN1) SPAN Y FINAL
     50 IF ((YN1-YF)*(YF-YN)) 60,70,110
C       (YN1,YN) DOES NOT SPAN YF, SET (XN,YN)=(XN1,YN1), REPEAT
     60 YN=YN1
        XN=XN1
        GOTO 170
C       EITHER YN OR YN1 =YF. CHECK WHICH AND RETURN PROPER (X,Y)
     70 IF (YN1-YF) 80,100,80
     80 ANSY=YN
        ANSX=XN
        GOTO 160
    100 ANSY=YN1
        ANSX=XN1
        GOTO 160
C       (YN,YN1) SPANS YF. TRY TO FIND X VALUE ASSOCIATED WITH YF
    110 DO 140 I=1,10
C       INTERPOLATE TO FIND NEW TIME STEP AND INTEGRATE ONE STEP
C       TRY TEN INTERPOLATIONS AT MOST
        HNEW=((YF-YN)/(YN1-YN))*(XN1-XN)
        JUMP=3
        GOTO 170
    115 XNEW=XX
        YNEW=YY
C       COMPARE COMPUTED Y VALUE WITH YF AND BRANCH
        IF (YNEW-YF) 120,150,130
C       ADVANCE, YF IS BETWEEN YNEW AND YN1
    120 YN=YNEW
        XN=XNEW
        GOTO 140
C       ADVANCE, YF IS BETWEEN YN AND YNEW
    130 YN1=YNEW
        XN1=XNEW
    140 CONTINUE
C       RETURN (XNEW,YF) AS ANSWER
    150 ANSX=XNEW
        ANSY=YF
    160 RETURN
C
    170 H2=HNEW/2.0
        T1=HNEW*FUN(XN,YN)
        T2=HNEW*FUN(XN+H2,YN+T1/2.0)
        T3=HNEW*FUN(XN+H2,YN+T2/2.0)
        T4=HNEW*FUN(XN+HNEW,YN+T3)
        YY=YN+(T1+2.0*T2+2.0*T3+T4)/6.0
        XX=XN+HNEW
        GOTO (25,45,115),JUMP
        END
```

Figure VIII-2: The SSP program RK1.

```
cond: init  IER=0
      split pred: split1 IF(XF-XI)11,11,12
                  split2 IF(HI) ,14,
      act1  ANSX=XI, ANSY=YI
      act2  IER=1, ANSX=XI, ANSY=0.0
      act3  cond: init  cond: init  H=HI
                              split IF(HI)16,,20
                              act1  H=-HI
                        XN=XI, YN=YI, HNEW=H
                  split temp comp: aug1  XN1=XN+HNEW
                                   term1 IF(XN1-XF)50,,
                                   aug2  YN1=integrate(HNEW,XN,YN)
                                   term2 IF((YN1-YF)*(YF-YN))60,,
                  act1  cond: split IF(XN1-XF) ,30,40
                              act1  ANSX=XF, ANSY=YN1
                              act2  HNEW=XF-YN, ANSX=XN+HNEW,
                                    ANSY=integrate(HNEW,XN,YN)
                              join  at label 160
                  act2  cond: split pred: split1 IF((YN1-YF)*(YF-YN)) ,70,110
                                          split2 IF(YN1-YF)80,100,80
                              act1  ANSY=YN, ANSX=XN
                              act2  ANSY=YN1, ANSX=XN1
                              act3  temp comp: aug1  DO 140 I=1,
                                               term1 DO 140    ,10
                                               aug2  HNEW=((YF-YN)/(YN1-YN))*(XN1-XN)
                                                     XNEW=XN+HNEW
                                                     YNEW=integrate(HNEW,XN,YN)
                                                     cond: split IF(YNEW-YF)120,,130
                                                           act1  XN=XNEW, YN=YNEW
                                                           act2  XN1=XNEW, YN1=YNEW
                                                           join  at label 140
                                               term2 IF (YNEW-YF)  ,150,
                              join  at label 160
                  join  at return
      join  at return
```

Figure VIII-3: A PBM analysis of RK1.

The mini-PA is designed to perform the same basic tasks as the full PA. Underlying all of its abilities is the ability to develop an understanding of a program by constructing a detailed plan for the program. The PA can use the plan as a basis for explaining the program in response to questions by the user. When the plan is being developed, the PA can detect bugs in the program by noticing inconsistencies in the evolving plan. When a bug is found, the PA can use the information in the plan in order to localize the source of the bug. Finally, the PA can use the plan in order to assess the ramifications of a proposed modification.

Consider how the mini-PA can determine a plan for a program such as RK1. The analysis component described in Chapter VI can determine a PBM plan for RK1 as shown in Figure VIII-3. Unfortunately, specification generating (as described in Chapter VII) cannot produce satisfactory specifications for a program which has as many levels of structure as RK1 has. In order to determine appropriate specifications, the mini-PA must rely on commentary by the programmer. Fortunately, excessively detailed annotation is not required. All that is needed is commentary which indicates the important features of specifications for some of the intermediate level segments. Specification generation can then be used to get the rest of the specifications. The amount of detail in the comments which actually appear in RK1 (see Figure VIII-2) is probably sufficient. (Note that the comments would have to be transformed into some machine understandable form.) The current design of the mini-PA and the full PA does not include the ability to understand or produce English. The system is intended to communicate with the user in some artificial language. This not withstanding, the examples below are given in English in order to make them more understandable to the reader.

The header comment for RK1, lines 1-26, gives the basic specifications for the program as a whole. The internal comments indicate features of the specifications for intermediate segments. For example, the comment on lines 58 & 59 indicates the specifications for lines 60-65, which implements action 2 of action 1 of action 3 of RK1. The comment also specifies the situation in which the segment will be executed. Similarly, the comments on lines 80, 82 & 83 give specifications for lines 81-160 which implements action 3 of action 2 of action 3 of RK1. It is assumed that the programmer would also provide some general knowledge such as the fact that the integral from A to B plus the integral from B to C is equal to the integral from A to C. Armed with specifications for some of the intermediate segments, the mini-PA can use specification generation in order to determine the specifications for the rest of the segments.

In those situations where specification generation is used to create the specifications for a segment, justification links supporting the specifications are produced as a part of the generation process. In those situations where the user provides specifications, the justification links have to be determined by the reasoning component. This is done by trying to show that the user's specifications follow from the specifications which would be generated for the segment. The weak reasoning component of the mini-PA is unlikely to be able to show that they do follow. When it cannot, it resorts to testing with examples. If it finds a counter example, a bug is reported as discussed below. Otherwise, the specifications are neither confirmed nor denied. The mini-PA simply accepts them provisionally as a basis for further specification generation and reasoning, while noting that they may in fact turn out to be incorrect at some future time.

An interesting aspect of the process of developing specifications for RK1 is that the kind of specifications which a programmer is likely to give for the program are inaccurate but never the less useful. For example, the most likely thing to say about RK1, and the computation on lines 104-109, is that they compute an integral. However, this is not true. They only approximate an integral. The inaccurate specifications are useful because they can be used as a basis for explaining most of the structure of the program. This is true because, for the most part, RK1 is written as if these specifications were true. The key thing which cannot be explained on the basis of these specifications is why the program uses a loop at all instead of performing one large step of integration. The PA can do useful things with inaccurate specifications because of the fact that it is designed with partial performance in mind. It does what it can with what it has.

Once a plan has been constructed, the PA can use it as a basis for describing the operation of the program. There are a wide variety of questions which a user can ask. They are all answered by reporting out features of the plan. Several examples showing possible questions, and how they might be answered by the mini-PA are given below. For example, the user could ask what some part of the program does.

> 1) What do lines 60-64 do?
> They compute ANSX=XF and ANSY=F(ANSX) starting with XN, YN, and FUN
> given that FUN(X,F(X))=dF/dx(X), and YN=F(XN).

This question is answered directly from the specifications of the corresponding segment in the plan.

A user could also ask a question about how parts of the program interact such as:

> 2) How do lines 60-64 work?
> The basic segment (lines 60-62) is a composition of line 60 (which determines HNEW so that XF=XN+HNEW) and lines 104-110 (which perform a step of integration yielding XX=XN+HNEW=XF and YY=F(XX)). Lines 61, 62, & 111 implement control flow links to lines 104-110. Lines (63,64) implement data flow passing the results along.

This is answered based directly on the data flow and control flow links in the plan, and the specifications of the subsegments of the segment in question. A user could also ask a question about the purpose of a part of the program such as:

> 3) Why are lines 60-64 in the program?
> The purpose of the lines is to compute ANSX and ANSY, two of the results of the program, in the situation where the loop implemented by lines 44-53 & 67-71 terminates with XN1>XF.

This is answered by looking at the supersegment of the segment in question in order to see how it is used. Finally, the user can ask questions about the logical structure of the program such as:

> 4) Why is the pre-condition of line 60 that XF be a floating number satisfied?
> Its satisfaction is guaranteed by a chain of pre-conditions reaching back to the pre-conditions of RK1. The value of XF is carried unchanged from an argument to the program RK1 which is required to be a floating number.
>
> 5) What justifies the post-condition of lines 60-64 that ANSX=XF?
> Data flow implemented by line 63 makes ANSX=XX. It is a post-condition of lines 104-110, the source of XX, that XX=XN+HNEW. It is a post-condition of line 60, the source of HNEW, that XN+HNEW=XF. Therefore, ANSX=XF.

The answers to these questions are based on the justification links in the plan. (Note that the justification links underlying the answer to question (5) illustrate what is probably as complex a deduction as the mini-PA is capable of.) It is easy for the mini-PA to answer questions like the ones above because the plan representation has been specifically designed to contain explicit information about all those topics.

One of the most useful tasks which the mini-PA can perform is bug detection and localization. As outlined above, the programmer's annotation is incorporated into the specifications of the segments in the plan. Differences between the actual operation of the program and the operation intended by the programmer are detected by the mini-PA when it tries to construct justification links summarizing proofs of correctness for each assertion in the specifications. The main tool it uses is testing each individual implication with examples. The full PA would be able to apply more powerful reasoning techniques to the task. The simple deduction component of the mini-PA is able to successfully find bugs in the program for two reasons. First, the hierarchical analysis of the program breaks the specifications up into a large number of small pieces which can be attacked separately. Second, most bugs in a program give rise to a number of problems in the program. A bug can be detected as long as one of the problems it causes can be detected.

As an example of bug detection, consider the program RK1. Inasmuch as it is a published program that people have been using for years, it has no bugs in it which can lead to catastrophic failure of the program. However, it does not operate in the manner that the comments clearly indicate that it should.

For example, an examination of the program readily shows that the comment in lines 20-21 does not correspond with the operation of the program. The comment clearly states that $XI \geq XF \rightarrow ANSY=YI$ and that $HI=0 \rightarrow ANSY=0.0$. The way the program operates ensures that $XI \geq XF \rightarrow ANSX=YI$ and that $ANSY=0.0$ only if $HI=0 \wedge XI<XF$. This bug can be detected based on the properties of the predicate which is used to select between the two actions. It can be seen that this is a bug in the comment, not in the program. The mode of operation indicated by the comment is impossible since $XI \geq XF$ and $HI=0$ are not mutually exclusive conditions. It is also clear that this is a minor bug, as long as some other programmer does not take the comment seriously and write another program which depends on the fact that $HI=0 \rightarrow ANSY=0.0$.

A more serious bug is involved with the search scheme embodied in lines 81-98. This segment uses repeated interpolations to seek out XNEW such that F(XNEW)=YF. It is an improved version of a halving search (in which line 84 would be HNEW=(XN1-XN)/2). The key to the method is that the interval (XN,XN1) always contains a root of F(X)-YF and that the size of the interval decreases at each step. A simple halving search is guaranteed to improve the accuracy of the result by a factor of 1024 in 10 iterations.

The search starts with YF in the interval (YN,YN1) (note the comment on line 80). A new value of Y is calculated (YNEW) and then the test in line 90 is used to determine whether YF is in the interval (YN,YNEW) or the interval (YNEW,YN1). This intention is indicated by the comments on lines 89, 91, & 95. The problem is that the particular predicate chosen to make the determination will work correctly only when YN<YN1. If YN>YN1, it consistently makes the wrong choice. As a result the search will only work in the intended fashion when YN<YN1. This, however, is not assured.

This bug can be detected by testing the segment on various values of YN and YN1. It is possible that a counter example will not be found, in which case the bug will go undetected. The full PA would probably have knowledge about binary searching in its plan library. If it did, it could detect the bug by recognizing the algorithm used here and knowing that it only works when YN<YN1.

The form of the test in line 67, which performs a similar function to the one on line 90, indicates that the programmer did not intend to limit RK1 to working on nondecreasing functions only. Therefore the test in line 90 is probably in error, and should be changed.

With this bug, the interval (XN,XN1) is no longer guaranteed to decrease in size, or to even contain a root of F(X)-YF. This would lead to considerable trouble if it were not for the fact that the search segment is robust. First, line 84 can extrapolate as well as interpolate. Second, the segment implemented by lines (104-110) will work fine with HNEW<0. Third, the segment implemented by lines (90-97) tends to create an interval (YN,YN1) containing a root of F(X)-YF where YN<YN1. As a result of this, after thrashing in the first couple of iterations, the search begins to work more or less correctly. This is probably why this bug was never found. Only rare pathological functions can cause catastrophic failure of the search.

This is an example of a definite bug that can be detected by comparing the operation of the program with the intentions of the programmer, but which is almost impossible to detect by looking at the performance of the program RK1 as a whole on test data.

Once a bug is detected, the PA localizes it to specific parts of the program by asking itself questions. It asks whether there was any justification for the claim which has just turned out to be false. If so, this justification is obviously spurious. The system tries to see whether any part of the justification has an obvious weakness. Depending on the type of error, the system determines what the other segments involved are. For instance, what the source of the variant input is or who will receive the variant output.

As an example, consider the second bug discovered above. After detecting this bug by finding a counter example to the pre-condition implied by the comment on line 91, the system might report the bug as follows:

> The comment on line 91 indicates that it is a pre-condition of the segment implemented by lines 92 & 93 that YF be between YNEW and YN1. A counter example has been found to this claim, namely: if YN=4, YF=3, YN1=1, and YNEW=2 then clearly YF=3 is between YN=4 and YN1=1, and YNEW=2 is less than YF=3, however, YF=3 is not between YNEW=2 and YN1=1.
>
> It should be noted that the test on line 90 tested only that YNEW<YF. The programmer implied that it could be proven that YF is between YNEW and YN1 whenever YF was between YN and YN1, and YNEW<YF. This is obviously not the case.

The system first describes the conflict which has arisen. Then it identifies the deduction which had been used to justify that XF would indeed be between YNEW and YN1. It should be noted that the mini-PA probably would not be able to develop such a pithy justification from the program as it stands. The mini-PA would start with a justification based on all of the pre-conditions of the segment on lines 81-98, all of the post-conditions of the segment on lines 84-88, and the post-conditions of line 90. This justification contains a lot of irrelevant information. In order to distill it down to a concise justification, the mini-PA would probably need assistance from the user. For instance, the mini-PA might ask, "Why does the fact that YNEW<YF from line 90 imply that YF is between YNEW and YN1, as required by line 91?" The user might answer "Because YF is between YN and YN1." The user could ask for more information if he desired, in order to decide whether the test on line 90 should be changed, or a prerequisite requiring that YN<YN1 should be added to the search.

A final ability of the mini-PA is its ability to understand the affects of a modification. There are two main aspects to modification: deletion and insertion. When something is deleted, the system must ask itself what depended on the existence of the deleted section, and therefore what loose ends have been left dangling. If there are no loose ends, then the section must not have been used for anything. When a section is inserted the system must ask itself two questions, "What contribution is this section going to make?" and "What problems is it going to cause?" The system can get hints about the answer to the first question by knowing that a section is replacing another, or supposedly fixing a bug. There are many potential problems that a modification may cause. Control or data flow paths may be disrupted. Needed assertions may be deleted or contradicted. This could cause justification links to become invalid. Consider some examples of how the system might respond to a suggested modification.

> 6) What would happen if line 21 were changed to
> "C OTHERWISE IF HI=ZERO THEN IER=1, ANSX=XI, AND ANSY=0.0"?
> This would eliminate a bug by removing the conflict between the old line 21 and the top level structure of the program.

Note, comments as well as code may be modified.

7) What if the test in line 90 were changed to "IF (YF-YNEW) 120,150,130"?
I (the system) assume you intend to claim that it is true that when XF is between YN and YN1 and YF<YNEW then YF is between YNEW and YN1. This is true for the example which brought up the bug. However, it also has counter-examples for instance, when YN=1, YF=2, YNEW=3 and YN1=4.

8) What if the test in line 90 is changed to
"IF ((YNEW-YF)*(YF-YN1)) 130,150,120"?
As far as I (the system) can tell, this fixes the bug without disrupting any of the other uses of line 90.

These questions are answered primarily with reference to the justification links in the plan. These links specify what depends on each part of the plan, and therefore what needs to be reverified when a change is made.

REFERENCES

[1] F.E. Allen and J. Cocke, "Theoretical Studies of Optimization", Proc. Symp. Compiler Optimization, SIGPLAN Notices, pp. 1-24, July 1970.

[2] F.E. Allen and J. Cocke, "A Program Data Flow Analysis Procedure", CACM V19 #3, pp. 137-147, March 1976.

[3] B.S. Baker, "An Algorithm for structuring Flowgraphs", JACM V24 #1, pp. 98-120, January 1977.

[4] R. Balzer, "Automatic Programming", Institute Technical Memo, University of Southern California / Information Sciences Institute, Los Angeles, CA, 1973.

[5] R. Balzer et. al., "Domain-Independent Automatic Programming", ISI/RR-73-14, University of Southern California, March 1974.

[6] D. Barstow, "Automatic Construction of Algorithms and Data Structures", PhD. Thesis, Stanford University, September 1977.

[7] D. Barstow and E. Kant, "Observations on The Interaction of Coding and Efficiency Knowledge in the PSI Program Synthesis System", Proceedings of The Second International Conference on Software Engineering, San Francisco, CA, pp. 19-31, October 1976.

[8] S. Basu and J. Misra, "Proving Loop Programs", IEEE Trans. on Software Eng. V1 #1, pp. 76-86, March 1975.

[9] S. Basu and J. Misra, "Some Classes of Naturally Provable Programs", Proc. 2nd Int. Conf. on Software Engineering, pp. 400-406, Oct 1976.

[10] M. Bauer, "A Basis for the Acquisition of Procedures From Protocols", IJCAI-75, USSR, 1975.

[11] A. Bawden, et.al., "LISP Machine Progress Report", MIT/AIM-444, August 1977.

[12] G.M. Birtwistle, O.-J. Dahl, B. Myhrhaug, and K. Nygaard, "SIMULA BEGIN", Auerbach, 1973

[13] R.S. Boyer and J.S. Moore, "Proving Theorems About LISP Functions", JACM V22 #1, January 1975.

[14] A.L. Brown, "Qualitative Knowledge, Causal Reasoning, and the Localization of Failures", MIT/AI/TR-362, March 1977.

[15] O.-J. Dahl and K. Nygaard, "SIMULA - An ALGOL-Based Simulation Language", CACM V9 #9, pp. 671-678, September 1966.

[16] O.-J. Dahl, E. Dijkstra, and C.A.R. Hoare, "Structured Programming", Academic Press, 1972.

[17] J. deKleer, "Local Methods for Localization of Faults in Electronic Circuits", MIT/AIM-394, 1976.

[18] J. deKleer, "A Theory of Plans for Electronic Circuits", MIT/AI/WP-144, May 1977.

[19] J. deKleer, J. Doyle, G. Steele, and G.J. Sussman, "AMORD: Explicit Control of Reasoning", Proceedings of the Symposium on Artificial Intelligence and Programming Languages, August 1977.

[20] J. Dennis, "First Version of a Data Flow Procedure Language" Project MAC Tech. Memo. #61, MIT Cambridge MA, May 1975.

[21] L.P. Deutsch, "An Interactive Program Verifier", PhD. Thesis University of California at Berkeley, June 1973.

[22] E.W. Dijkstra, "A Discipline of Programming", Prentice-Hall, Englewood Cliffs, N.J., 1976.

[23] T.A. Dolotta and J.R. Mashey, "An Introduction to the Programmer's Workbench", Proceedings of the Second International Conference on Software Engineering, San Francisco CA, pp. 164-168, October 1976.

[24] V. Donzea-Gouge, G. Huet, G. Kahn, B. Lang, and J.J. Levy, "A Structure-Oriented Program Editor: A First Step Towards Computer Assisted Programming", Report 114, Institut de Recherche en Informatique et Automatique, France, 1975.

[25] J. Doyle, "Truth Maintenance Systems for Problem Solving", MIT/AI/TR-419, 1977.

[26] R.W. Floyd, "Assigning Meaning to Programs", proc. symp. in Applied Math. V19, pp. 19-32, 1967.

[27] R.W. Floyd, "Toward Interactive Design of Correct Programs", IFIP, 1971.

[28] L.D. Fosdick and L.J. Osterweil, "Data Flow Analysis In Software Reliability", ACM Computing Surveys V8 #3, pp. 305-330, September 1976.

[29] D.P. Friedman and D.S. Wise, "Cons Should Not Evaluate its Arguments", Indiana Technical Report 44, November 1975.

[30] M.R. Genesereth, "Automated Consultation for Complex Computer Systems", Ph.D. Thesis Harvard Univ., August 1978.

[31] S.L. Gerhart, "Knowledge About Programs: A Model and Case Study", SIGPLAN Notices V10, #6, Proceedings of the International Conference on Reliable Software, 1975.

[32] S.L. Gerhart, "Correctness-Preserving Program Transformations", Proc. of 2nd Symp. on Principles of Programming Languages, Palo Alto, 1975.

[33] S. German and B. Wegbreit, "A Synthesizer of Inductive Assertions", IEEE Trans. on Software Eng. V1 #1, pp. 68-75, March 1975.

[34] I.P. Goldstein, "Understanding Simple Picture Programs", MIT/AI/TR-294, MIT Cambridge MA, September 1974.

[35] I.P. Goldstein and M.L. Miller, "Structured Planning and Debugging, A Linguistic Theory of Design", MIT/AIM-387, December 1976.

[36] D.I. Good, "Provable Programming", ACM SIGPLAN Notices V10 #6, Proc. of International Conf. on Reliable Software, 1975.

[37] G.C. Green, "The Design of The PSI Program Synthesis System", Proceedings of The Second International Conference on Software Engineering, San Francisco CA, pp. 4-18, October 1976.

[38] G.C. Green, "A Summary of The PSI Program Synthesis System", IJCAI-77, Cambridge MA, pp. 380-381, August 1977.

[39] G.C. Green and D.R. Barstow, "Some Rules for the Automatic Synthesis of Programs", IJCAI-75, USSR, September 1975.

[40] J.V. Guttag, "The Specification and Application to Programming of Abstract Data Types", Technical Report CSRG-59, University of Toronto, September 1975.

[41] M. Hammer, W.G. Howe, V.J. Kruskal, and I. Wladawsky, "A Very High Level Programming Language for Data Processing Applications", IBM research report RC-5583, Yorktown Heights N.Y., 1975.

[42] S. Hardy, "Synthesis of LISP Functions From Examples", IJCAI-75, USSR, 1975.

[43] P. Henderson and J.H. Morris, "A Lazy Evaluator" SIGPLAN-SIGACT POPL Conf., Atlanta, January 1976.

[44] C. Hewitt and B. Smith, "Towards a Programming Apprentice", IEEE Trans. on Software Eng. V1 #1, pp. 26-46, March 1975.

[45] C.A.R. Hoare, "An Axiomatic basis for Computer Programming", CACM V12 #10, pp. 576-583, October 1969.

[46] C.A.R. Hoare, "Proof of Correctness of Data Representations", Acta Informatica V1 #4, pp. 271-281, 1972.

[47] C.A.R. Hoare and N. Wirth, "An Axiomatic Definition of the Programming Language PASCAL", Acta Informatica V2 #4, pp. 335-355, 1973.

[48] IBM C28-6515-6, "FORTRAN IV Language", IBM White Plains NY, 1966.

[49] IBM GH20-0205-4, "Scientific Subroutine Package Version III Programmer's Manual", IBM White Plains NY, 1970.

[50] S. Igarashi, R. London, and D. Luckham, "Automatic Program Verification I: A Logical Basis and Its Implementation", Stanford AIM-200, May 1973.

[51] G. Kahn, "The Semantics of a Simple Language for Parallel Programming", IFIP-74, North-Holland Publ. Co, 1974.

[52] G.Kahn and D.B. MacQueen, "Coroutines and Networks of Parallel Processes", IFIP-77, North-Holland Publ. Co, 1977.

[53] E. Kant, "The Selection of Efficient Implementations for a High Level Language", Proceedings of the Symposium on Artificial Intelligence and Programming Languages, Rochester NY, August 1977.

[54] S.M. Katz, and Z. Manna, "A Heuristic Approach to Program Verification", IJCAI-73, Stanford Univ., 1973.

[55] S. Katz and Z. Manna, "Logical Analysis of Programs", CACM V19 #4, pp. 188-206, April 1976.

[56] J. King, "A Program Verifier", PhD. Thesis, Carnegie Mellon University, 1969.

[57] J. King, "Proving Programs to be Correct", IEEE Trans. on Computers V20 #11, November 1971.

[58] D.E. Knuth, "An Empirical Study of FORTRAN Programs", Software Practice and Experience V1 #2, pp. 105-134, 1971.

[59] D.E. Knuth, "The Art of Computer Programming", Volumes 1-3, Addison-Wesley, 1968, 1969, 1973.

[60] B. Liskov, "A Note on CLU", MIT/Computation Structures Group Memo 112, MIT/LCS, November 1974.

[61] B.H. Liskov, A. Snyder, R. Atkinson, and C. Schaffert, "Abstraction Mechanisms in CLU", CACM V20 #8, pp. 564-576, August 1977.

[62] S.D. Litvintchouk and V.R. Pratt, "A Proof-Checker for Dynamic Logic", MIT/AIM-429, June 1977.

[63] R. London, "A View of Program Verification", ACM SIGPLAN Notices V10 #6, Proc. of International Conf. on Reliable Software, 1975.

[64] Z. Manna and A. Pnueli, "Axiomatic Approach to Total Correctness of Programs", in Acta Informatica V3, pp. 253-263, 1974.

[65] Z. Manna and R. Waldinger, "Knowledge and Reasoning in Program Synthesis", Artificial Intelligence V6, pp. 175-208, 1975.

[66] Z. Manna and R. Waldinger, "Is 'sometime' Sometimes Better Than 'always'? Intermittent Assertions In Proving Program Correctness", Proc. 2nd Int. Conf. on Software Engineering, October 1976.

[67] Z. Manna and R. Waldinger, "Synthesis: Dreams => Programs", Stanford Research Institute Technical Note 156, November 1977.

[68] M.L. Miller and I.P. Goldstein "SPADE: A Grammar Based Editor For Planning and Debugging Programs", MIT/AIM-386, December 1976.

[69] M.L. Miller and I.P. Goldstein "Structured Planning and Debugging", IJCAI-77, August 1977.

[70] D.A. Moon, "MACLISP Reference Manual", MIT Cambridge MA, 1974.

[71] J.S. Moore, "Introducing PROG into the PURE LISP Theorem Prover", Xerox PARC Report CSL-74-3, 1974.

[72] J.H. Morris jr. and B. Wegbreit, "Subgoal Induction", CACM V20 #4, pp. 209-222, April 1977.

[73] L.J. Osterweil, and L.D. Fosdick, "DAVE - a FORTRAN Program Analysis System", Proc. Computer Science and Statistics: 8th Annual Symposium on the Interface, pp. 329-335, 1975.

[74] L.J. Osterweil, and L.D. Fosdick, "DAVE - a Validation, Error Detection and Documentation System for FORTRAN Programs", Software Practice and Experience, 1976.

[75] M.R. Paige, "On Partitioning Program Graphs", IEEE Trans. on Software Eng., V3 #6, November 1977.

[76] R.P. Polivka and S. Pakin, "APL: The Language and its Usage", Prentice-Hall, Englewood cliffs NJ, 1975.

[77] T.W. Pratt, "Control Computations and the Design of Loop Control Structures", IEEE Trans. on Software Eng., V4 #2, pp. 81-89, March 1978.

[78] V. Pratt, "Semantical Considerations on Floyd-Hoare Logic", MIT/LCS/TR-168, September 1976.

[79] C. Rich and H. Shrobe, "Initial Report on a LISP Programmer's Apprentice", MIT/AI-TR-354, MIT Cambridge MA, December 1976.

[80] C. Rich, "Plan Recognition In A Programmer's Apprentice", MIT/AI/WP-147, May 1977.

[81] C. Rich, H.E. Shrobe, R.C. Waters, G.J. Sussman, and C.E. Hewitt, "Programming Viewed as an Engineering Activity", MIT/AIM-459, MIT Cambridge MA, January 1978.

[82] C. Rich, "A Representation for Programming Knowledge and Applications to Recognition, Generation, and Cataloging of Programs", forthcoming PhD thesis, MIT.

[83] G. Ruth, "Analysis of Algorithm Implementations", MIT PhD Thesis, MAC/TR-130, 1973.

[84] G. Ruth, "Protosystem I: An Automatic Programming System Prototype", MIT/LCS/TM-72, 1976.

[85] G. Ruth, "Automatic Design of Data Processing Systems", 23rd ACM POPL Conf., 1976.

[86] E.D. Sacerdoti, "The Non-Linear Nature of Plans", SRI AI Technical Note 101, 1975.

[87] J.T. Schwartz, "On Programming, An Interim Report on the SETL Project; Installment 1: Generalities", Courant Institute of Mathematical Sciences, New York University, February 1973.

[88] J.T. Schwartz, "On Correct Program Technology", in Courant Computer Science Report #12, September 1977.

[89] D. Shaw, W. Swartout, and C. Green "Inferring LISP Programs from Examples", IJCAI-75, USSR, 1975.

[90] M. Shaw and W. A. Wulf, "Abstraction and Verification in ALPHARD: Defining and Specifying Iteration and Generators", CACM V20 #8, pp. 553-564, August 1977.

[91] H.E. Shrobe, "Plan Verification in A Programmer's Apprentice", MIT/AI/WP-158, January 1978.

[92] H.E. Shrobe, "Reasoning and Logic for Complex Program Understanding", PhD thesis, MIT, August 1978.

[93] B. Smith, R. Waters, and H. Lieberman, "Comments on Comments or the Purpose of Intentions, and the Intentions of Purposes", a paper written as part of the MIT course "Automating Knowledge Based Programming and Validation Using Actors", taught by C. Hewitt, Fall 1973.

[94] R. Stallman and G.J. Sussman, "Forward Reasoning and Dependency-Directed Backtracking In A System for Computer-Aided Circuit Analysis", Artificial Intelligence Journal, October 1977.

[95] P.D. Summers, "Program Construction From Examples", PhD thesis Yale Univ., 1975.

[96] G. Sussman, "A Computational Model of Skill Acquisition", MIT/AI/TR-297, MIT Cambridge MA, August 1973.

[97] G.J. Sussman and R. Stallman, "Heuristic Techniques in Computer Aided Circuit Analysis", IEEE Transactions on Circuits and Systems V22 #11, November 1975.

[98] G.J. Sussman and G.L. Steele jr, "Scheme an Interpreter for Extended Lambda Calculus", MIT/AIM-349, December 1975.

[99] G.J. Sussman, "SLICES: At The Boundary Between Analysis and Synthesis", MIT AIM-433, July 1977.

[100] N. Suzuki, "Automatic Verification of Programs with Complex Data Structures", Stanford AIM-279, February 1976.

[101] W. Teitelman, "Automatic Programming - The Programmer's Assistant", Proceedings of the 1972 FJCC, pp. 917-921, 1972.

[102] W. Teitelman, "INTERLISP Reference Manual", XEROX, Palo Alto CA, 1974.

[103] W. Teitelman, "A Display Oriented Programmer's Assistant", IJCAI-77, August 1977.

[104] R. Waldinger and K.N. Levitt, "Reasoning About Programs", Artificial Intelligence V5, pp. 235-316, 1974.

[105] R.C. Waters, "A System for Understanding Mathematical FORTRAN Programs", MIT/AIM-368, MIT Cambridge MA, August 1976.

[106] R.C. Waters, "A Method, Based on Plans, for Understanding How a Loop Implements a Computation", MIT/AI/WP-150, July 1977.

[107] R.C. Waters, "A Method For Automatically Analyzing the Logical Structure of Programs", MIT PhD. Thesis, August 1978.

[108] R.C. Waters, "A Method For Analyzing Loop Programs", to appear in IEEE Trans. on Soft. Eng., May 1979.

[109] B. Wegbreit, "Heuristic Methods for Mechanically Deriving Inductive Assertions", IJCAI-73, Stanford Univ., 1973.

[110] B. Wegbreit, "The Synthesis of Loop Predicates", CACM V17 #2, pp. 102-112, February 1974.

[111] T. Winograd, "Breaking the Complexity Barrier Again", Proceedings of the ACM SIGIR-SIGPLAN Interface Meeting, November 1973.

[112] W.A. Wulf, "ALPHARD: Towards a Language to Support Structured Programming", Carnegie Mellon Univ., 1974.

[113] W.A. Wulf, R. London, and M. Shaw, "An Introduction to the Construction and Verification of Alphard Programs", IEEE Transactions on Software Engineering V2 #4, December 1976.

[114] A. Yonezawa, "Symbolic Evaluation Using Conceptual Representations For Programs With Side-Effects", MIT/AIM-399, December 1976.

[115] A. Yonezawa, "Verification and specification Techniques for Parallel Programs based on Message-Passing Semantics", MIT PhD thesis, December 1977.

[116] S. Zilles, "Abstract Specification for Data Types", IBM Research Laboratory, San Jose California, 1975.